Winter 2012

Implementation of an RDMA verbs driver for GridFTP

Timothy Joseph Carlin
University of New Hampshire, Durham

IMPLEMENTATION OF AN RDMA VERBS DRIVER FOR GRIDFTP

BY

Timothy Joseph Carlin

B.S., University of New Hampshire (2006)

THESIS

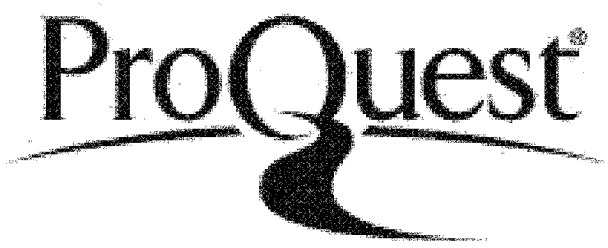
Submitted to the University of New Hampshire
in Partial Fulfillment of
the Requirements for the Degree of

Master of Science

in

Computer Science

December, 2012



ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

This thesis has been examined and approved.

Thesis director, Robert Russell
Associate Professor of Computer Science

Radim Bartoš,
Associate Professor & Chair of Computer Science

Robert Noseworthy
UNH-IOL, Chief Engineer

28 Nov 2012
Date

DEDICATION

For my family who are near and those who are far, those who are here in person, and for those who are here in spirit.

ACKNOWLEDGMENTS

I would like to first thank Dr. Robert D. Russell for giving me the opportunity to join him in his research, and for leading me to this thesis. Also, for all of his help and guidance throughout the process, and for all of the time spent discussing and reviewing my work. His detailed comments and insight were invaluable.

I would also like to thank the staff and students at the University of New Hampshire InterOperability Lab (UNH-IOL) for their continued support and extensive technical knowledge. I would like to also thank the OpenFabrics Alliance (OFA) for their support.

Lastly, I would like to acknowledge and thank Rajkumar Kettimuthu and Michael Link, both of Argonne National Laboratory, for their help with the Globus XIO code base. There were many times where their expertise helped reduce even the most daunting of challenges to minor problems.

This research used resources of the ESnet Testbed, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-05CH11231.

This material is based upon work supported by the National Science Foundation under Grant No. OCI-1127228.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

IMPLEMENTATION

OF AN

RDMA VERBS DRIVER

FOR GRIDFTP

by

Timothy Joseph Carlin

University of New Hampshire, December, 2012

GridFTP is used by researchers to move large data-sets across grid networks. Its benefits include data striping, multiple parallel data channels, and check-pointing. Like traditional FTP, GridFTP separates the control channel from the data channel, but also includes extensions to allow for third-party control of the transfer. GridFTP is one component of the Globus Toolkit, a software system designed to allow researchers to easily adapt, extend, and modify GridFTP and other tools to meet their specific needs. The eXtensible Input/Output (XIO) component, which provides common and logging libraries as well as built-in transport drivers such as TCP and UDP, defines the architecture and API used for implementing new drivers.

Remote Direct Memory Access, or RDMA, is a network communication architecture which defines a number of features not available with traditional sockets based communication. With RDMA all I/O operations are offloaded to hardware. Combined with the use of registered memory, this eliminates the intermediate copying of data to kernel buffers as is done with TCP. This results in a significant reduction in latency and CPU overhead because the kernel TCP/IP stack is not used. The RDMA architecture is implemented with InfiniBand cables and interface cards, or with Ethernet cables and RoCE or iWARP interface cards.

This thesis funded in part by National Science Foundation Grant OCI-1127228, and describes the implementation of a new Globus Toolkit XIO driver which uses the RDMA Verbs API to drive network communication. Unlike other XIO drivers, the RDMA driver will not produce an intermediate copy of data prior to transmission over the network, therefore taking advantage of the full capabilities of RDMA. The work presented here describes the motivation, implementation, and various challenges and solutions associated with implementing a new XIO driver for RDMA, as well as giving an overview of its effectiveness when compared with TCP.

CHAPTER 1

Introduction

1.1 Introduction

The increasing power of computer systems to process data quickly and efficiently has had a wide reaching impact across many industries. In the scientific community, these advances have a much broader impact. With faster and more simultaneously available processors comes the ability to run more complex experiments and simulations capable of generating tremendous amounts of data, many hundreds of gigabytes, terabytes, or more.

The motivation for moving data is clear. Research may be collaborative, with laboratories in separate locations each with a desire to process the data set independently. Another reason may be data redundancy. Data storage devices are always vulnerable to accidental deletion, corruption, or destruction by natural disaster. An off site, redundant copy helps ensure that any undesired loss can be recovered. Another example can simply be to move the data to a location better suited for processing. The systems responsible for generating or collecting data may be simple, automated devices, and potentially in remote locations, whereas the systems responsible for processing the data may be very complex. A related situation is one where data collected from separate locations is meaningless, unless brought together for processing. Perhaps the best example of these factors is the world-wide team of physicists and engineers working on data produced by the Large Hadron Collider (LHC). Located on the border of France and Switzerland in Europe, experiments performed with the LHC are generating huge amounts of data that requires the expertise of scientists located all over the world.

1.2 Motivation

There have been numerous and significant advances on a number of fronts to increase the performance of data transfers over wide-area grid networks. These advances include the availability of 100GbE, rate-guaranteed virtual circuits, and finely tuned TCP [24] and UDP [23] transport protocols. To accompany these advances, researchers have developed tools to match, namely GridFTP.

There still exists an opportunity to improve on the performance of these networks. For large file transfers, variability in the performance has been seen throughout the duration of transfers. The National Center for Atmospheric Research participated in a study which exhibits this variability. Over a period of 4 months, 120TB of data was transferred using GridFTP. Rates of 70-200MB/sec were seen, over a 10Gb/s capable link, and these rates varied greatly. [32]

1.3 GridFTP and RoCE

This work seeks to identify RDMA, and in particular, RoCE as a means of obtaining consistent throughput, ease of set-up, and high bandwidth utilization with GridFTP.

GridFTP[10] is a protocol designed to make the data transfer process in grid computing environments more efficient, reliable, and faster. Built upon traditional FTP[25], GridFTP offers several desirable enhancements and configuration options, including performance tuning, striping, and a 3rd party connection controller. In this thesis, the term GridFTP also refers to the Globus Toolkit application which implements the GridFTP protocol [12].

RDMA[26] is an alternative network communication protocol which offers a number of advantages over traditional network interfaces. In particular, RDMA operates asynchronously, offloads processing to dedicated hardware, and reduces the number of copies required to move data. The results are: lower overall CPU usage, less memory overhead, and higher throughput.

RDMA over Converged Ethernet (RoCE) is a data transfer protocol which allows RDMA

verbs to be used over an Ethernet (802.3) connection. While RoCE was designed for Converged Ethernet, it has been shown to be effective over traditional Ethernet on links which are not congested. This means RoCE will allow users to take advantage of RDMA protocols, while continuing to use pre-existing Ethernet interconnections.

1.4 Benefits

There are at least four benefits that may be realized from the outcome of this work.

With traditional TCP sockets-based network communication, data is copied several times between the time it is read from a local disk, and written to a remote disk. The copies that take place on the source node are as follows: 1) from Disk to Application Memory, 2) Application Memory to the Network Buffer, and 3) from Network Buffer to the Network. This process is reversed and repeated at the destination node. This process is CPU intensive, as well being burdensome to memory resources and memory bus bandwidth. It is in reducing these copies, then, where the most dramatic benefits are realized.

The first benefit of utilizing RDMA, and specifically InfiniBand or RoCE, comes simply from not using TCP. TCP accounts for CPU overhead for the Operating System to maintain the connection state, resources, and to execute the transfer protocol. This benefit will be realized simply by using the proposed RDMA library for communication, without modification to any existing application.

The rest of the benefits outlined deal directly with how memory buffers are created and maintained throughout the life of the application. To allow Channel Adapters access to application memory, applications need to register the memory region in which data buffers will exist. The process of registering memory is done to allow the channel adapter to read directly from the application's virtual memory space, eliminating the need for a context switch and also eliminating the copy from user memory to a network buffer.

Data read from disk into application memory can then be pushed directly to the network by the channel adapter. This results in a further significant reduction of CPU utilization.

Memory considerations are also the subject of the next two benefits. The first con-

sideration comes from the overall memory usage. Since data isn't copied to and from an intermediate buffer, the intermediate buffer need not be allocated. In the case of large buffers (needed to achieve high bandwidth utilization) this results in significant memory savings. Second, in the case of TCP, applications in user-space are each vying for the services of the operating system to establish connections and transfer data. When large amounts of data are being transferred, there is contention for memory bandwidth. This results in a memory bottleneck for applications.

1.5 Requirements

There are several requirements that must be met for the RDMA driver to be best utilized by applications.

1.5.1 Integration

The first set of requirements deals with the seamless integration of the driver. The XIO portion of the Globus Toolkit allows for any driver to be loaded and used "on-the-fly" without any modification of application code. This is a feature which must carry over to the RDMA driver. GridFTP is an established tool, one in which patches and updates can not be taken lightly, without close scrutiny. Lastly, all features must be available, and operate within prescribed parameters, without modifying the application.

1.5.2 Synchronization

RDMA is by its very nature an asynchronous architecture. This paradigm must carry over to applications using it. By carrying over this behavior, applications can choose to take advantage of it or not, without driver imposed limitation. The XIO API requires that a minimal set of functions be implemented for every driver. This minimal set includes: open, close, read, and write. Each function has two modes: a non-registered mode allowing for serial operation, and a registered mode allowing for asynchronous operation. GridFTP makes use of the registered functions for its I/O operations.

1.5.3 Memory

The last requirement deals with memory and buffer usage. As previously discussed, RDMA Channel Adapters require that memory be registered in order to gain access for zero-copy transfer. At start-up most RDMA applications will register the memory regions from which data buffers are created. However, no such registration method currently exists in GridFTP. There are a number of alternative methods available for registering memory that will be investigated.

1. Data buffers can be registered and deregistered as they are passed between application and driver. This eliminates any overhead introduced by intermediate copying buffers of data to registered memory, and reduces complexity with memory registration management, however this will dramatically increase the number of registration/deregistration operations.

2. Data buffers can be registered as control is transferred to the driver. Rather than deregistering the buffers, memory registrations are instead cached, and when a buffer is seen again, its previous registration is reused. This "on-the-fly" registration, however, can only be effective if the following two conditions are met. First, the buffers must be reused within some consistent interval. Without reuse, at some point, the entirety of memory will be registered, over-subscribing the application's quota. Second, the buffers must be relatively small. Even with reuse, extremely large buffers will again quickly result in over-subscribed memory allocation. This method, while complex, reduces the overall number of registration and deregistration operations, and also requires no intermediate copying.

3. Data buffers can be copied into and out of driver registered memory. This requires that the driver allocate and register a region of memory large enough to handle the user data. While reducing complexity, and the number of registration and deregistrations, this introduces an intermediate copying of the data. This method is functionally similar to the behavior of TCP/IP sockets in that respect.

4. An interface can be provided to GridFTP, and a modification to the code made which allows GridFTP to register the region of memory from which buffers will be allocated. This method is the most desirable method, however due to the GridFTP changes required, is more complicated to implement and make available to users.

GridFTP uses relatively small buffers, by default 256KiB in size, and reuses buffers. The size of the buffers can also be configured via a command line switch. Future work may involve an update to GridFTP to introduce a call to register the memory region from which buffers are allocated, to reduce the memory caching costs.

1.6 Outline

This work outlines the efforts necessary to implement an RDMA transport driver for the GridFTP software. The performance of transfers using traditional GridFTP TCP/IP based drivers, and the new driver using RDMA Verbs will also be compared.

The rest of this paper is organized as follows:

- Chapter Two gives background on RDMA
- Chapter Three gives background on the Globus Project, including GridFTP
- Chapter Four gives an overview of related work
- Chapter Five describes the GridFTP software architecture, and gives details showing how the RDMA driver is implemented.
- Chapter Six describes the experiments showing the performance of GridFTP with both traditional, and RDMA drivers.
- Chapter Seven analyzes the results of testing
- Chapter Eight summarizes the work shown here as well as suggesting avenues for further research.

CHAPTER 2

RDMA

2.1 RDMA

RDMA [26] stands for "Remote Direct Memory Access". A parsing of the name reveals with good accuracy what the protocol allows: RMDA gives direct access to the memory of a remote node by a local node. The "directness" of RDMA comes from the fact that data is transferred memory to memory, without unnecessary intermediate copying. Data is transferred directly from a user buffer on the source node directly to a user buffer on the destination node. In contrast, traditional sockets copy data from a user buffer to a kernel buffer on the source node side, and again, in reverse, from the kernel buffer to a user buffer on the destination node. [18]

2.2 OFED

The OpenFabrics Alliance (OFA) develops, tests, licenses, supports and distributes OpenFabrics Enterprise Distribution (OFED^{TM}) open source software to deliver high-efficiency computing, wire-speed messaging, ultra-low microsecond latencies and fast I/O. [9] OFED supports InfiniBand, iWARP, and RoCE on both Windows and GNU/Linux based systems. The API provided by OFED is known as the Verbs API.

All communication functions of RDMA protocols are offloaded to the respective Channel Adapters of RoCE, InfiniBand, and iWARP. This hardware offloading allows for extremely low latency network I/O, low CPU utilization, and reduced memory bottlenecks. User-level applications communicate directly with the Channel Adapter, and completely bypass operating system calls for data transfer operations. The tradeoff of this approach is a

more complex code architecture, API, and other runtime requirements, such as memory registration, not normally seen with traditional socket based I/O.

2.3 Transports

The InfiniBand architecture [7] upon which RDMA is based is no different from many other data communication technologies in that it uses a version of the OSI "Layer" model to separate functionality. In the layer model, lower layers deliver data, and provide services, for higher layers. In general, higher layers tend to be more complex, and more specific to a given task, for example, HTTP is a protocol specific to viewing web content. InfiniBand provides distinct implementations for each layer in the architecture.

Three RDMA technologies are currently available: 1) InfiniBand, 2) iWARP, and 3) RDMA over Converged Ethernet, or RoCE. RoCE allows RDMA to operate over a traditional, or converged Ethernet network. Unlike iWARP, RoCE does not incur penalties due to IP or TCP operation, as it utilizes InfiniBand's network and transport layers, while using Ethernet's physical and link layer.

2.3.1 InfiniBand

When viewed from the Application, InfiniBand provides a channel to allow direct reading and writing of remote memory. To do this, InfiniBand uses unique hardware, called a Host Channel Adapter (HCA). In addition to representing the Physical Layer of the protocol stack, the HCA also implements most of the functionality of the remaining layers, which is what allows for all I/O transfer processing to be free from the system CPU.

The InfiniBand Link Layer can be likened to Ethernet and is used to establish connections between HCAs. An InfiniBand Network layer is described, however as yet is unimplemented, and not used. The lack of a strict network layer means that InfiniBand cannot yet be routed in the same way an IP network is, and instead is switched, more like Ethernet networks. Unlike Ethernet, InfiniBand does not allow broadcast transmission in the traditional Ethernet sense, however it does provide a multicast facility, similar to that

used in IP.

Like TCP, a Transport Layer is defined to establish connections between applications using InfiniBand services. The Transport layer protocols are defined according to delivery guarantees: reliable or unreliable and connected or datagram transport. Reliable, connected services can be seen as similar to TCP[24], while unreliable services are similar to UDP[23]. The transport layer also provides functionality for atomic network operations, and multicast delivery, among others. [17]

2.3.2 RoCE

RoCE [8] is based on the InfiniBand architecture, and differs only in the implementation of the lowest two layers, the link and physical layers, where RoCE utilizes the corresponding Ethernet layers. Although a software based solution known as "Soft RoCE" is available, RoCE is generally implemented in an Ethernet channel adapter, and for all work conducted in this research 10Gb/s Channel Adapters were used.

The RoCE Channel Adapter is known as as RoCE Network Interface Card (RoCE "NIC") and operates in the same way as other RDMA Channel adapters.

It's important to note that the top two layers of the InfiniBand architecture (the transport and network layers) are identical in RoCE and InfiniBand. Therefore the same user-level services are provided in both technologies, namely reliable and unreliable service, connected and unconnected service, atomic operations over the network, multicast, etc. This also means that software written to use the OFA InfiniBand API will execute without modification using either protocol (this is mostly true for iWARP as well).

While RoCE is intended for, and will enjoy the benefits provided by Converged Ethernet, it can operate on any Ethernet LAN. Converged Ethernet refers to the lossless guarantees provided when a network utilizes the IEEE Data Center Bridging Standards (802.1Qau, 802.1Qaz, and 802.1Qbb). Without the lossless guarantee, RoCE performance is expected to degrade on congested links. This independence from Converged Ethernet results in a simpler network design and infrastructure, allowing for a wider range of devices that may

not yet implement the necessary standards.

Like InfiniBand, RoCE has no routing capabilities akin to IP Routing. The lack of a routing function is due to the use of the InfiniBand network layer, which is currently defined as an "empty" layer, which runs directly above Ethernet.

RoCE provides low-latency, high-bandwidth RDMA communication between hosts that are connected to the same Ethernet broadcast domain. Due to this restriction, both InfiniBand and RoCE are suited for use as the interconnection between nodes in a data center. In a data center environment, nodes are generally located in close physical proximity, and higher layer routing is generally unnecessary.

In this work however, it is shown how RoCE can be used across a long distance network, while still maintaining a high level of throughput, albeit with higher latency. To do this, IEEE 802.1Q VLANs can be utilized with MPLS virtual circuits to connect remote LANs.

2.3.3 iWARP

iWARP [28][26] is very different from the other RDMA technologies. Unlike InfiniBand and RoCE, iWARP utilizes the common TCP/IP/Ethernet infrastructure for all data transfer operations. This makes iWARP the only RDMA technology capable of operating natively on IP-routed networks. As with RoCE and InfiniBand, the operations of iWARP can be offloaded to an RNIC, allowing for the host operating system to be bypassed, and thus achieving the same low-latency, low-CPU utilization operation.

While iWARP benefits from the services provided by IP and Routing, it is also limited by the same. Since it runs over TCP, it is bound by the same rules of operation, namely, the back-off and restart algorithms TCP uses in the face of congestion. Also, since iWARP isn't isolated to a single LAN, it can't take full advantage of Converged Ethernet, unless all LANs in the path utilize Converged Ethernet. iWARP performance will be similar to RoCE on a converged network, though iWARP will have extra overhead requirements due to TCP. In a lossy environment, RoCE will break, though thanks to TCP iWARP will sustain its integrity.

2.4 Transfer Semantics

RDMA provides two different semantics for transferring data. The semantics provided are known as Send/Receive, or channel semantics, and RDMA Write/RDMA Read, or memory semantics. These methods of I/O have no strong correlation to the built-in models of any other protocol.

2.4.1 RDMA Send/Receive

In the Send/Receive semantics, work requests are posted to the work queue of both the requesting and responding channel adapters. The requesting side posts the work request to the SEND queue, while the responder posts a work request to the RECEIVE queue. The Send/Receive semantics require that both sides of the connection take an active role in the data transfer. Each Send/Receive operation must be explicitly accounted and prepared for by both parties. Using this mode, the receiving node is notified immediately when a data transfer operation has completed, and is able to take action.

2.4.2 RDMA Read/RDMA Write

The RDMA Write and RDMA Read are one-sided operations where semantics require no active involvement by the responding party. The requesting party uses the RDMA Write to directly push data, from the requesting channel adapter, to the responding channel adapter.

To do this, an RDMA Write work request is posted to the send queue. No work request is required on the responding side, and no work completion will be posted on the responding side. Alternatively, a requesting party may use the RDMA Read operation to pull data from the responding channel adapter to the requesting channel adapter. Again, no work request need be posted on the responder, and none will be fulfilled.

The RDMA Write, RDMA Read semantics allows for one side of the connection to remain completely passive, while the other is actively transferring data. To use RDMA Write and RDMA Read, the node responsible for moving the data must know the address of the remote memory location to write to (or read from). A single Send/Receive exchange

is commonly used to deliver this information prior to the RDMA Write or RDMA Read.

A "hybrid" mode exists, known as RDMA Write with immediate. With this mode the remote device is notified when the RDMA Write operation is completed. The amount of "immediate" data that can be delivered is limited to 4 bytes.

CHAPTER 3

Globus

3.1 Overview

The Globus Alliance is an international consortium of educational and scientific institutions collaborating to advance science and technology through the use of Grid Computing.

The Globus Alliance is headquartered at Argonne National Laboratory, and its members include the University of Southern California's Information Sciences Institute, the University of Chicago, the University of Edinburgh, the Swedish Center for Parallel Computers, and the National Center for Supercomputing Applications (NCSA). The ways the Globus Alliance works to advance the Grid Computing are varied. The alliance creates, or helps to create, a wide variety of software to utilize the grid, as well as to build and support the community and infrastructure involved in the development and use of the software. In addition, it generates rich documentation and standards governing the operation of tools and infrastructure. [15]

3.2 Toolkit

The Globus Toolkit (GT) is a software architecture providing services to more easily take advantage of the capabilities of grid computing [3]. The Globus Toolkit is an open source project allowing for, and encouraging, extensions and improvements to be made. The components provided by the Globus Toolkit are divided into Security, Data Management, Execution Management, and Common Runtime categories. GridFTP is part of the Data Management features, while the eXtensible IO (XIO) driver architecture used by GridFTP is part of the Common Runtime.

Globus XIO [11] provides an abstract I/O system to the Globus Toolkit. The intent is to keep the operations of applications using the Toolkit from the details associated with the transfer of data. This also allows different protocols to be substituted with minimal, if any, changes to the application itself.

In aiding new developers in the task of creating new drivers, several template drivers, examples, and skeleton code files are available.

3.3 GridFTP

GridFTP is a protocol designed for moving data across the grid, and the application of the same name that implements the protocol. There are a number of considerations when dealing with grid computing that GridFTP strives to address.

3.3.1 Protocol

Using the term "grid" implies several uniquely identifying characteristics. A Grid would not have a centralized controller, instead, nodes belonging to a grid would operate independently, but in concert, with an agreed upon set of policies. Further, the policies and protocols in use over a Grid are standard, and open. Without this openness, the Grid devolves into segmented applications, rather than a harmonious blend of nodes available for all tasks. Lastly, it is expected that a Grid will have some guarantees as to the availability, quality, and speed of services. A grid which does not take into account steps to provide this again becomes segmented, with each part less capable than the grid as a whole. [16]

The GridFTP protocol is an open and freely available standard. The features GridFTP provides can be categorized according to how they contribute to the transfer.

Data striping and parallel transfers are two features GridFTP provides that deal directly with the movement of data. In most cases where GridFTP is used, the data set to be moved is very large, either in number or size of files, or both. Data striping allows different portions of a single file to be transported between various network interfaces in parallel. Dividing the file among various independent transfers allows any and all network bandwidth available

between two nodes to be maximized. In addition, so as not to lose the work done in setting up the transfers, GridFTP has the ability to mark checkpoints. The checkpoints are used to identify when a file has been partially transferred so upon restart the transfer may continue from the same point.

The ability to control data through a third party client is a key component to the GridFTP architecture. This allows for each node of the Grid to run an instance of a GridFTP Server, and for a GridFTP client to initiate and control the transfer from another location. As the control channel will require far less bandwidth than the transfer itself, the requirements for the client are far less restrictive. Other applications of this feature stretch to utilizing the Internet, and dedicated Web Services for orchestrating data transfer, such as Globus Online [2]. A diagram of this interaction is shown in fig. 3-1. The third-party controller also is used to give desired parameters of the transfer to the servers, such as the number of parallel streams, TCP buffer and window sizes, and security.

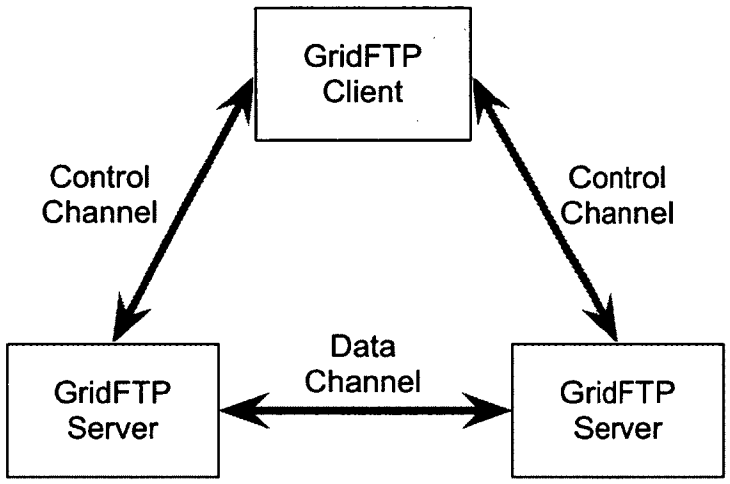


Figure 3-1: GridFTP 3^{rd} Party Control

3.3.2 Application

The GridFTP application itself is split up into a number of different executable programs. The two primary applications used by this work are *globus-gridftp-server (ggs)* and *globus-url-copy (guc)*. *globus-gridftp-server* serves as the server application meant to run on each grid node capable of moving data. Accordingly, *globus-url-copy* is the client program, capable of initiating a local data transfer, or controlling a transfer between two servers.

3.4 XIO

The Globus eXtensible Input/Output (XIO) library is an abstraction of the set of system calls necessary for I/O operations. The goal of XIO is to provide a unified interface to the application, allowing any protocol to be used "under-the-hood" to perform the transfer. This removes any dependency on specific libraries or implementation decisions and allows the developer to concentrate on the application. The only API the developer need be aware of is a simple and straightforward Open/Close/Read/Write I/O API. In addition, this allows for changes in the I/O scheme to be made without modification of the application. These changes can be to take advantage of new hardware, adding security in the form of encryption or integrity, or for logging or auditing, or some combination of features.

3.4.1 Driver

The abstraction of the XIO architecture allows for data transfer protocols to be swapped out simply, as they all follow the same API. Components implementing the XIO data transfer API are known as "drivers". Drivers are loaded dynamically, as needed by the application. As they are loaded, a logical stack of drivers is built, with processing starting at the top-most driver, and flowing down through each driver in order, until the bottom-most driver copies data to the final destination (e.g. the network device). This process is shown in fig. 3-2. Drivers that interface directly with the hardware are known as "transport" drivers. There can be only one transport driver on a stack, and it must be on the bottom. Drivers which

are not transport drivers are "transform" drivers. These drivers may modify, augment, or audit the data they receive prior to passing it to the next driver.

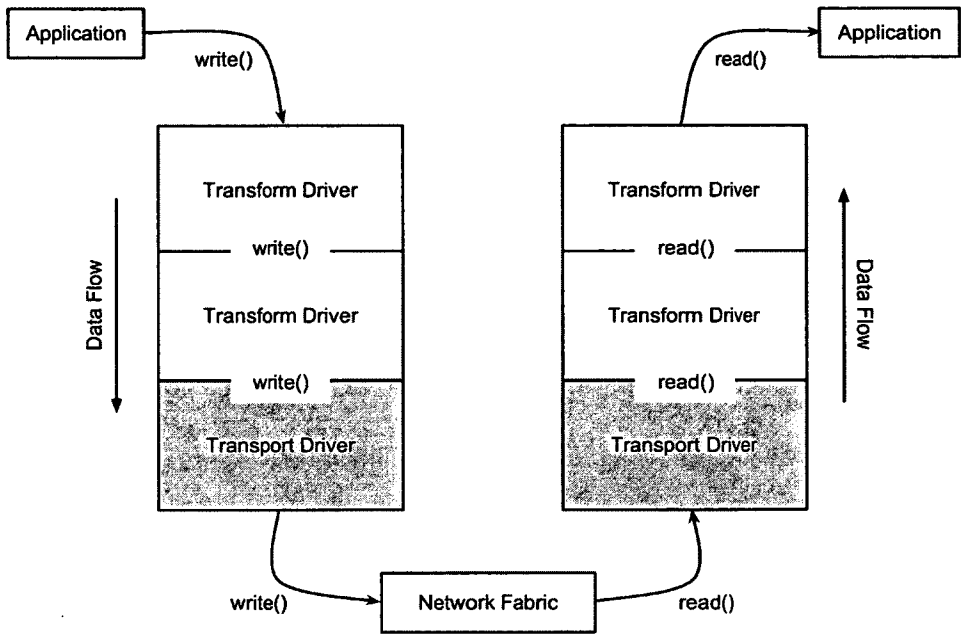


Figure 3-2: Globus XIO Driver Stack

3.4.2 Asynchronous API

The XIO library allows for all I/O operations to be run in either a synchronous, or asynchronous mode. In the synchronous mode, I/O operations are executed inline, blocking on each call. In the asynchronous mode, operations are scheduled. This allows the application to complete other work while waiting to be notified that a transfer has completed. Both versions are made available for every driver API. In fact, a driver implementation does not consider in any way what mode is being utilized. The only requirement is that when an operation completes, an associated "finished" call is made by the driver to indicate the status to the application.

CHAPTER 4

Related Work

This chapter reviews previous work involving the use of RDMA across a distance and other approaches to defining RDMA capabilities for GridFTP.

4.1 RDMA Methods

The results learned from [22] will be of significant importance. RDMA has a number of complex, powerful, and flexible features available to it. This work examines each of the features individually and gives performance results for various combinations. This data is used in the RDMA driver implementation to focus on factors which will be most important to the overall performance, while de-emphasizing those which have little effect.

4.2 RDMA over Distance

RDMA is traditionally thought of as a protocol isolated to the confines of a machine room or data center. Indeed, some of RDMA's primary benefits of a low latency, flat network, and direct access to memory are features well-suited for a single administrative domain, with machines in close proximity. Further, the hardware requirements of RDMA, and in particular InfiniBand, limit the ability of separately managed data centers to readily connect. However, research is showing that, with the appropriate device configuration and software techniques, not only can RDMA be used effectively over large distances, but with very high efficiency.

4.2.1 InfiniBand over Distance

InfiniBand was designed for use inside of a data center, and as such, expects a very small propagation delay between ports of a subnet. This requirement prevents a network designer from simply extending the length of the connecting cable to join two geographically separated networks.

To alleviate this requirement and allow InfiniBand connections to achieve much greater distances new hardware, known as Range Extenders, are introduced. Also known as InfiniBand over Wide-Area (IBoWA), range extenders are deployed in pairs, and available from various vendors [6] [1]. To "extend" the range of InfiniBand, each range extender acts as either an InfiniBand switch or router for the "Local" side of the connection. Data is then transferred using SONET to the remote range extender, where the process is reversed.

With the physical problem of extending the range of InfiniBand solved, the question then becomes: How well does it work? It has been shown [14] that InfiniBand performs well as the transport method for a clustered file system for a 1 PetaFLOP/s super computer. The Lustre [5] file system was used to store data. RDMA, IP over InfiniBand (IPoIB), and Sockets Direct Protocol (SDP) were each tested in both local-area and wide area test cases. The results indicate that over distances of 1400, 6600, and 8600 miles, that InfiniBand scales well. These results also show that when using the Reliable-Connected mode of InfiniBand larger message sizes perform far better than smaller messages.

InfiniBand over large distances has also shown to have good performance in a production environment. Using the Lustre file system and IBM Bladecenter nodes, an InfiniBand interconnection was made over a range of 28km [27], also using Obsidian Longbow network adapters.

4.2.2 RoCE over Distance

RoCE over distance has an advantage over InfiniBand in that no extra translation of the protocol is needed to extend the range of transmission. Since RoCE is able to use a standard Ethernet connection, the only extra infrastructure needed is a RoCE capable card.

RoCE does impose some restrictions on the nature of the network. First, as RDMA runs directly over Ethernet, both endpoints must reside in the same Ethernet broadcast domain, or the same LAN. Remote LANs may be connected using an IEEE 802.1Q VLAN, or other protocols. The second restriction imposed by RoCE is the requirement of a converged Ethernet. Unlike InfiniBand, standard Ethernet is lossy, which means that at any time a packet may be corrupted, truncated, or simply not delivered. When the set of IEEE 802.1 Data Center Bridging standards are applied to all segments of the Ethernet, it is considered converged, and makes a guarantee to be lossless. RoCE will benefit from a lossless network, as it will have a guarantee that all data is delivered.

Previous work [31] [13] has shown that despite distance between the nodes, a consistently high throughput can be achieved. This is a surprising, yet welcome, outcome. RDMA, and InfiniBand in particular, is traditionally isolated to a local data center environment, where the physical distance between devices is small. Using RoCE over the ANI testbed, it has been shown that with enough buffers, data can be transferred with better than 95% link utilization.

The latency observed with the transfer is proportional to the distance between nodes which is true of any transfer protocol. In addition, though not created for high-latency networks, the RoCE protocol operates correctly, without exhibiting any data loss or corruption, or protocol errors.

In order to achieve high throughput, the network link, and subsequently the Channel Adapter, must be kept full and active at all times. This translates to allocating a number of buffers which decreases with the size of the data portions transferred. This means that RDMA, and in particular, RoCE is a feasible choice for a GridFTP driver.

It is also important to note that very low CPU usage was maintained for the duration of the transfers. It should be also be noted that most testing has been done with memory-to-memory transfers, so as to not incur penalties due to disk interaction. Also, in the face of contention for the link with other protocols such as TCP, RoCE is able to maintain good performance.

4.3 RDMA for GridFTP

There have been a number of prior and concurrent explorations into adding RDMA capabilities to Globus XIO and GridFTP in particular. Distinguishing these methods are the model used for transferring data, the libraries used, as well as hardware augmentations. In this section, we explore several of these alternative approaches.

4.3.1 ADTS

The Advanced Data Transfer Service (ADTS) was introduced to provide a zero-copy FTP [25] for bulk data transfers across a WAN [21]. ADTS implements a library of services which use RDMA to to provide a transfer layer and optimized transfer interfaces. In addition, an FTP wrapper interface to ADTS was implemented. In the third phase, a Globus XIO driver using ADTS was designed and implemented [29]. Of particular note is the implementation of the ADTS library itself. Of the two options available for transferring data, Channel Semantics, using RDMA Send/Recv were chosen over Memory Semantics, using RDMA Read or RDMA Write. There are several reasons given to justify this approach. Channel Semantics do not require the writer to have any information about the remote buffer info, eliminating the need for an explicit advertisement. In addition, no acknowledgment that the transfer has completed is necessary with Channel Semantics, as the completion of the Recv operation serves as notification. In the case of RDMA Write, the "reading" node must have some awareness of a completed operation. (The RDMA_WRITE_WITH_IMM operation delivers an explicit completion event to the reader when the write has completed.) The ADTS driver utilizes a circular buffer, as well as a read thread and a write thread, as part of the implementation. Buffers locations are pre-fetched, so as to be loaded and ready when the network is available.

One drawback of the approach of ADTS is the exclusive use of a circular buffer to queue data for I/O operations. As described, the implementation induces an extra copy of each data segment at both the transmitting and receiving node as data is moved to and from the circular buffer. This means that RDMA's zero-copy capability is not fully exercised,

leading to increased latency and CPU utilization.

4.3.2 RXIO

Another possibility for adding RDMA capabilities to the Globus Toolkit is explored with RXIO [30]. Rather than implementing a driver for XIO, the researchers chose to extend, and augment the XIO architecture itself. It is well documented that many of the semantics and methods for using RDMA represent a stark departure from those of TCP sockets. In adapting XIO for RDMA, the researchers sought to make XIO more readily integrate with RDMA. The resulting implementation could then be used with an RDMA driver, or any other driver not using RDMA Verbs, such as TCP or UDT. Two protocols were established, one for the transfer of short messages and the other to transport large messages. Registering data buffers for transfer is not an inexpensive process, so the decision to be made is if copying a buffer to a pre-registered buffer would be faster than registering the same buffer. For buffers large enough to justify registration, a "rendezvous" protocol is introduced. With this protocol, an exchange takes place between sender and receiver to learn information necessary for a subsequent RDMA_WRITE operation.

4.3.3 SLaBS RDMA Driver

The authors of [19] take another approach to an RDMA driver for Globus XIO using previous results from work done on the Phoebus system [20]. The Phoebus system introduces a hardware gateway to the edge of the network. The gateway dynamically reserves resources and does load-balancing, as well as making application specific adjustments to improve throughput and overall performance. In doing this, the Phoebus system breaks end-to-end connectivity, enabling it to manage individual connections directly. One of the outputs of the earlier work is the development of the Session Layer Burst Switching Protocol (SLaBS). The notion of SLaBS is to create compartments, or slabs of data, to transfer in bulk over the network. In developing the RDMA Driver, the authors used the SLaBS buffer model to build data units for transmission. Of particular note here is the use of the RDMA_READ

operation for transferring data. The RDMA_READ operation is initiated by the destination server, and "pulls" data from the source server, where RDMA_WRITE "pushes" data from source to destination.

CHAPTER 5

Implementation

5.1 Overview

Each driver in the XIO architecture must implement a minimum set of functions. XIO allows for client and server functions to be defined, as well as a set of attribute functions. Open, close, read, and write functions must be specified at a minimum. Server operation additionally requires init, accept, and destroy functions to be implemented. The XIO architecture allows for both asynchronous and synchronous functions to be defined.

The RDMA driver specifies asynchronous versions of each operation. Also specified is the opaque handle data-type which is used to set various options and store information associated with a connection. The handle is an argument for each I/O operation, in the same fashion as a socket descriptor.

5.2 Driver Activate/Init

The driver activate and init functions serve to initialize the driver functions with the XIO subsystem and are called to enable a driver within a stack. The *globus_xio_driver_set_transport* function is used by the driver to set the I/O functions, as well as to classify it as a transport driver. The red shaded blocks in fig. 3-2 indicates where the RDMA driver must fall in the GridFTP stack. As a transport driver it must be on the bottom to interface directly with network hardware.

In this case of *globus-url-copy* and *globus-gridftp-server*, these functions are called at start-up, when the data stack is set. With *globus-url-copy* the driver stack is set by the user when the application is run. The *globus-gridftp-server* is generally started at boot-time,

and may participate in many different transfers. Also, not all drivers may be appropriate for all servers. For instance, the RDMA driver would not be used on a server without RDMA hardware. To accommodate this, in *globus-gridftp-server* a set of drivers must be "whitelisted", or permitted to be used. If a driver is not whitelisted, it will not be used, and an error will be issued to the client.

The "activate" function in particular merely registers the existence of the driver. The driver "init" function, not to be confused with the server init function, is called to initialize the driver jump tables. It is in the init function where the type of the driver is set, as well as pointers to the various driver functions and capabilities. In the case of the RDMA driver, transport, server, attribute, and options function pointers are declared.

5.3 Read/Write Functions

The driver read and write functions are called after the client and server sides have rendezvoused and both open functions have completed operation. At this point, all attributes have been set, and both reader and writer have fully configured handles.

An array of *iovec* structures is used to deliver data to and from the read and write functions. These "scatter-gather" elements, allow for multiple data-buffers, of potentially different sizes, to used in a single I/O operation.

It is the writer that determines how the transfer will proceed. The reader simply waits for an indication from the writer which mode is being utilized for the transfer. There are two modes available to the RDMA driver. In the first mode, known as "Buffer Copy Mode", the driver will copy data in the *struct iovec* array to a private buffer space. In the second mode, known as "Buffer Register Mode", the driver utilizes the *iovec* buffers directly, without any copying. These methods are detailed in later sections.

At the completion of a successful write operation, the number of bytes written is signaled to the calling function. In the event that not all of the data in the *struct iovec* was written, this number may be less than the total size of the *struct iovec* array. This may happen due to mismatched writer and reader buffer sizes, or when the final buffer of data is transferred.

When no further data is to be transferred, the driver receives a *struct iovec* array of size 1, with the first element having an *iov_len* equal to 0. When this EOF has been reached, the writer sends a "finished" message to the reader to signal the completion of the transfer. At this point the writer exits successfully, having written no further data.

5.3.1 Buffer Copy Mode

In "buffer copy mode", the writer and reader use a private buffer space that was previously initialized and registered during the open. Each block that is written is first copied to an intermediate buffer, before being copied to the network device. In the case of the read function, the data is copied into an intermediate buffer from the network device, before being copied into the user buffer space. In this mode, the operation closely resembles how the common sockets API operates. This mode does not take advantage of the ability of RDMA to avoid extra copying of data, however it does avoid the need to register user buffers dynamically. Figure 5-1 shows how data moves through the GridFTP application to the RDMA Driver, and finally to the Channel Adapter. The left side of the diagram shows how the data is copied to a private buffer space which is already associated with registrations. The right side of the diagram shows the buffer register mode of operation where data buffers are registered, or a prior registration is retrieved, prior to being sent to the Channel Adapter. Buffer register mode is discussed in more detail in the next section.

In the copy mode, since private buffer space is being utilized, the size of the source and destination buffers is a known, agreed upon value. This allows the transfer to proceed without any concern as to the available space in the reader buffer.

The RDMA_WRITE_WITH_IMM RDMA operation is used to move data across the network. When this operation completes, the reader is notified that the write has been completed with the "immediate data". The immediate data is available to the writer to specify any arbitrary data, up to 4 bytes. This data is used in this implementation to indicate to the reader the last buffer written to by the sender. An internal counter is also maintained by the reader to indicate the last buffer returned to the user. This allows for the

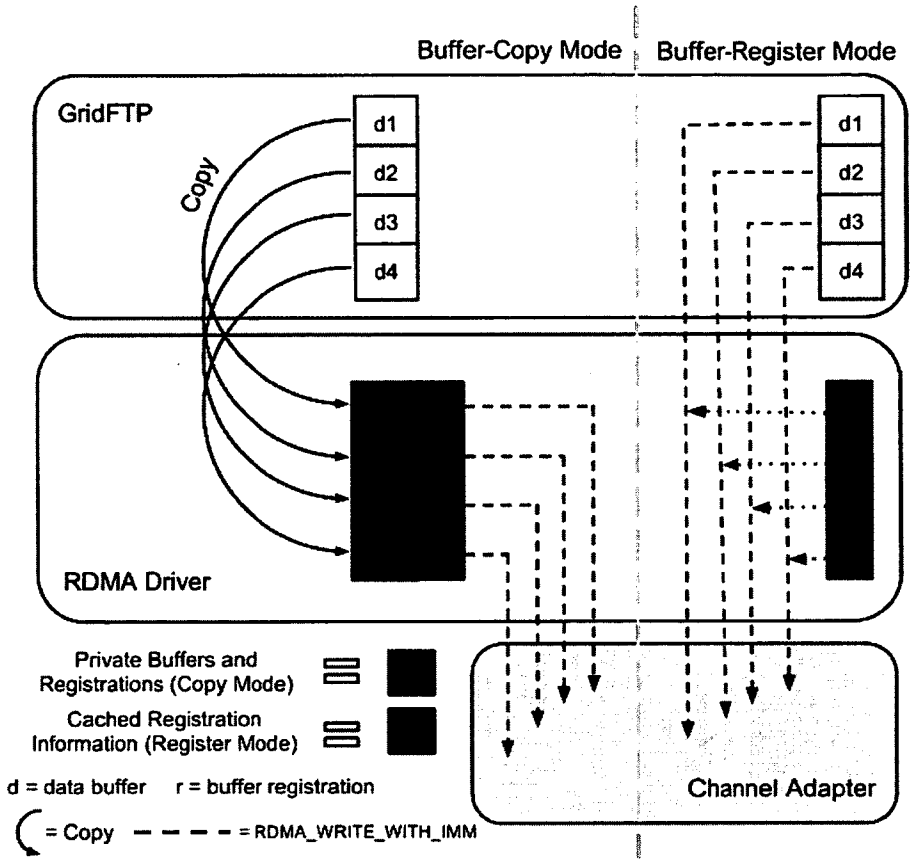


Figure 5-1: Movement of Data Buffers with RDMA Driver

sender to transmit data to the private buffer space ahead of what might have been read by the user. The immediate data is delivered with an explicit work completion, which notifies the reader that the write has finished and the amount of data written. This allows for partially-filled buffers in the private space. The full read/write protocol is detailed below.

5.3.2 Buffer Register Mode

In "buffer register mode" the full capabilities of RDMA are emphasized. The driver will not copy the data to an intermediate destination. Instead, data will be sent directly to the network from the user buffer. In doing this, the write function must register the user buffer prior to transmitting data. Along with the registration, a "send work request" is created for that buffer. This "on-the-fly" buffer registration is then stored in the handle so that

future writes may reuse the registration. Registrations are cached by storing the registration information in a linked-list keyed off of the base address of the memory region. This is not an overly efficient method for caching the registration, however, tests have shown that two alternating data buffers are used to deliver data to the driver for I/O. If at some future time a method is implemented to allow the user to specify the number of buffers used in I/O operations, then this choice of a linked-list for caching should be revisited. By caching the registrations, the process of re-registering the same buffer is avoided, and the process of de-registering buffers is postponed to the close function. Likewise, the read function will register and cache the user buffers given to it, and data will be written directly to these buffers. The reader needs only to signal to the caller the number of bytes actually written to the buffer.

To accomplish direct memory-to-memory transfers the writer must receive information from the reader about the remote buffers. The information that must be delivered includes the address of the buffer, the remote access key, and the buffer length. This information is included with each send work-request. The impact of this is an extra exchange that must take place prior to every registered exchange, adding time and complexity. The full read/write protocol is detailed below.

Directly utilizing the user's buffers for the transfer adds the possible complication of mismatched read and write buffer lengths. There are three cases to examine. The case of the read buffers being larger than the write user buffers needs no special handling. The completion generated at the receiver by the RDMA_WRITE_WITH_IMM operation specifies how much data was written. Since the data is placed in the user buffers directly via the channel adapter, all that is left is to set the *iovec.iov_len* value to the actual length read, resulting in partially filled buffers. The case where the read and write buffers are equal is also handled by this case.

The case where the write buffer is larger than the read buffer needs special consideration. With this case, it becomes necessary for the writer to not overwrite the reader buffer. What follows is the unusual case of the number of written bytes being less than the number of

bytes offered to the write function. With the sockets API, or any API utilizing internal buffering, this is not generally the case. Even though data may not be immediately written, rather than return fewer written bytes, pending data is moved up in the circular buffer. With the RDMA driver this data is not buffered and the caller must retry the write of the remaining data.

5.3.3 Read/Write Protocol

The tests described in the Results Chapter used the pseudo-code algorithms described in this section to transfer data. The SEND and RECV operations represent the RDMA Send/Recv operations. For example, *RECV mode* will wait for a corresponding *SEND*, and store the received data in the *mode* variable. The WAIT operation indicates that the code is waiting for a completion event, as in the one that would be issued in on the target node of an RDMA_WRITE_WITH_IMM. The messages exchanged by the driver in copy mode and register mode are depicted in fig. 5-2 and fig. 5-3, respectively.

Write Algorithm

```
Write(struct iovec data_buffers[ ])
  if data_buffers[0].iov_len == 0 then
    SEND( EOF )
    signal SUCCESS, 0 bytes written
    return
  end if
  if any buffer in data_buffers > ONE_MB then
    let buffer_register = true
  else
    let buffer_register = false
  end if
  if buffer_register then
    for buffer in data_buffers do
      register or retrieve registration
    end for
    SEND( use_register_mode and num_buffers )
    RECV( num_buffers )
    RECV( bufferinformation )
    for buffer in data_buffers do
      RDMA_WRITE_WITH_IMM( buffer )
    end for
  else
    SEND( use_copy_mode and num_buffers )
    RECV( num_buffers )
    for buffer in data_buffers do
      copy buffer to private_buffer
      RDMA_WRITE_WITH_IMM( private_buffer )
    end for
  end if
  signal SUCCESS, N bytes written
  return
```

Read Algorithm

```
Read(struct iovec data_buffers[ ])
  RECV( mode )
  if mode == EOF then
    signal SUCCESS, EOF
    return
  end if
  if buffer_register then
    for buffer in data_buffers do
      register or retrieve registration
    end for
    SEND( num_buffers )
    SEND( data_buffers.registration_information )
    for buffer in data_buffers do
      WAIT write completion
    end for
  else
    SEND( num_buffers )
    for buffer in data_buffers do
      WAIT write completion
      copy buffer to private_buffer
    end for
  end if
  signal SUCCESS, N bytes read
  return
```

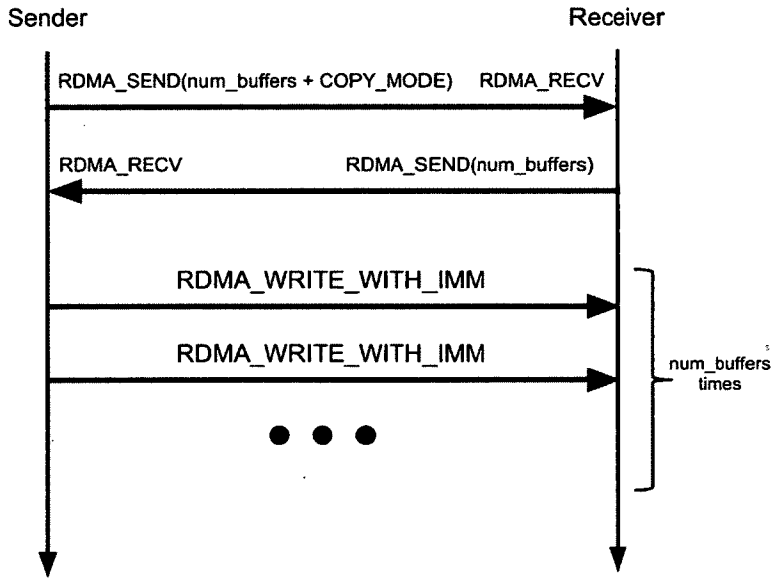


Figure 5-2: Message Exchange - Copy Mode

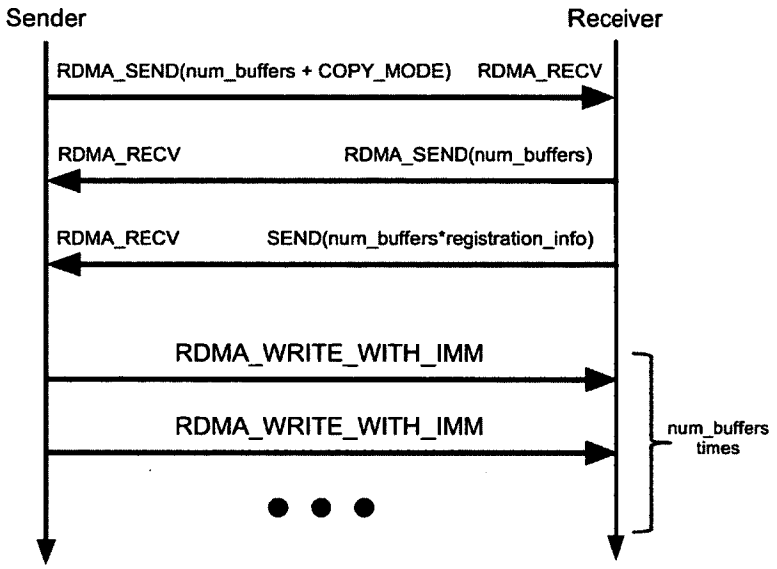


Figure 5-3: Message Exchange - Register Mode

5.4 Open/Close Functions

The open and close functions connect the reader and writer, teardown the connection, setup the private data structures in the driver specific handle, as well as configuring any requested attributes.

The client side of the connection, which becomes the writer, is delivered the contact information of the server by the calling function. The contact information is collected by the caller of the *server_init* function, which in the case of GridFTP is *globus_url_copy*. *globus_url_copy* uses the information for itself in the case of a direct transfer, or relays it to another server instance in the case of a mediated transfer.

The open function is also responsible for allocating, configuring, and exchanging information about the private buffer space. The private buffers are used by the "buffer copy method" of the write and read operations. The reader and writer exchange the location, length, and remote access key for each of the private buffers.

In the case of the reader, the work of the setting up the driver handle is done by the server accept function. The only functionality of the open function utilized by the reading side of the connection is the configuration of any user-requested driver attributes.

5.5 Server Specific

Prior to an exchange using the RDMA driver there is no distinction between a writer/reader or client/server. Before a transfer can take place, one side must assume the role of the server. There are two functions that must be called to enable server operations, these functions "init" and "accept" are detailed below.

5.5.1 Init

The job of the server init function is to establish an RDMA listener. The interface and port may be configured through attributes, otherwise default parameters will be used. Prior to exiting, the server init function notifies the caller of the contact information to be used by a future client.

5.5.2 Accept

The server accept function is called immediately following the completion of server init. Like a traditional socket accept, the RDMA driver accept will block until a connection is made. When a rendezvous is made an "agent", or reader, handle is established. The same work accomplished by the client open function is done by the accept, including the exchange of private buffer information. Upon the completion of the accept, the new agent handle is fully initialized to transfer data.

5.6 Attributes

The RDMA Driver allows for several attributes to be configured. These attributes can be directly set and retrieved by the user and are the user's hook into the internal workings of the driver. Attributes must be configured prior to the opening of a connection. This work uses the attributes to configure the server interface, port, and the number and size of the private buffer space. The mode of the transfer, buffer copy or register, may also be selected through the attributes.

CHAPTER 6

Results

This chapter discusses a number of tests which were run in the process of validating the RDMA Driver, as well as documents the results and data obtained. Also discussed is the test environment itself.

6.1 Test Setup

The following tests were run between two identical nodes, each consisting of twin 6-core Intel Westmere-EP 2.93GHz processors with 6 GB of RAM and PCIe-2.0x8 running OFED 1.5.4 on Scientific Linux 6.1. Each node has a dual-port Mellanox MT26428 adapter with 256 byte cache line, one port configured for InfiniBand 4xQDR ConnectX VPI with 4096-byte MTUs, the other for 10 Gbps Ethernet with 9000-byte jumbo frames. With this configuration, each RoCE frame can carry up to 4096 bytes of user data. All transfers use Reliable Connection (RC) transport mode which fragments and reassembles large user messages. Nodes are each connected inline to an ANUE Systems XGEM traffic shaper. The ANUE System is used to introduce delay on the line, to simulate latency experienced by transfers over a long distance.

6.1.1 GridFTP Options

There are several options available to GridFTP users that were used to execute the various tests. These options are described below.

- Parallel Connections

  The "-p" option to *globus-url-copy* specifies the number of parallel connections, or streams, opened between the two servers to perform the transfer. This should not

be confused with the number of server threads. In order to compare the RDMA and TCP drivers, values of 1, 2, and 4 connections were tested, where the recommended value for TCP is between 4-8 connections.

- TCP Block Size

  When using the TCP driver, the TCP block size can be set via the "-tcp-bs" option to *globus-url-copy*. This represents the amount of memory allocated by the kernel, and the amount of data that can be sent before receiving an acknowledgement. Therefore, this value is dependent on the bandwidth delay product, as well as the number of parallel transfers. The following formula recommended by Globus XIO developers was used in determining the value: $b * R * (1000/8/p - 1)$ [4], where b represents the bandwidth in Mb, R represents the round-trip time in ms, and p is the number of parallel connections. In the case of 1 connection, the formula was modified as follows to prevent division by zero: $b * R * (1000/8/\max(1, p - 1))$.

- Transfer Duration

  The "-t" option to *globus-url-copy* allows for transfers to continue for up to a given number of seconds.

- Read/Write Block Size

  The "-bs" option to *globus-gridftp-server* specifies the block size of data read from the file, and subsequently posted to the network. As RDMA is message-based, the block size plays an important role in the maximum throughput and this value was varied for each test.

- Server Threads

  The "-threads" option to *globus-gridftp-server* specifies the number of threads available to the server during execution. The recommended value for this is 1 or 2. For the purposes of this work, 1 thread delivered a reasonable level of performance, and was used throughout.

6.1.2 Test Scenarios

Using the previously specified program options, several different test scenarios were investigated. These tests were done to demonstrate differences in throughput and CPU utilization under various circumstances, and to highlight optimal parameters.

- Disk-to-Disk Transfers

  The servers used for testing were not suitably configured for high-performance disk I/O. As a result, the disk proved to be a bottleneck limiting the maximum throughput that either driver could achieve. These tests were performed to verify the integrity of the transferred data by validating the SHA-1 Checksum of the file on both the source and destination servers.

- Memory-to-Memory Transfers

  Memory-to-memory transfers were performed to eliminate the bottleneck caused by disk I/O. Data was read from the special "/dev/zero" file provided by the operating system, which generates as many bytes of zeroes as are read from it, without the possibility of failure. Data was written to the special "/dev/null" file provided by the operating system, which will always succeed, without writing any data to physical memory. The challenge with this transfer method is limiting the size of the transfer. With files stored on disk, end-of-file is indicated by a special end-of-file character. "/dev/zero", by definition, has no end-of-file. To limit the amount of data the GNU/Linux *dd* utility was used as an intermediary between the data source, "/dev/zero", and *globus-gridftp-server*. To do this, *dd* reads data from "/dev/zero" at constant 1KiB block sizes, up to a given number of bytes, and writes to a named FIFO, or pipe, created with the "mkfifo" command. With this approach, the normal *globus-gridftp-server* file interface could be used, with data read from memory. The drawback of this approach is two-fold. First, the use of *dd* necessitates an extra copy of data, to and from the FIFO, which will increase CPU utilization. The second limit is the size of the FIFO itself. On the systems used for testing, a named FIFO has a

size of 512 8 byte blocks, or 4096 bytes.

- Timed Transfers

  To see the effects of using the *dd* process, timed transfers were performed using the "-t" option. With this setup, data could be read directly from "/dev/zero", and again written to "/dev/null". No end-of-file is necessary, as the transfer will be cancelled after the transfer timer expires. While not realistic in a production environment, this mode is useful for isolating the capabilities of the drivers. Timed transfers were run for 5 minutes each (300 seconds, or "-t 300").

- Large File Transfers

  Extremely large files were transferred from memory to memory, using *dd* and the method described above. These tests were done to demonstrate the effects of any delay caused by start-up or teardown, as the cost is amortized over a longer period.

- Very Long Distance Transfers

  Transfers were also performed over very long distances. This was accomplished by increasing the delay induced on the line by the ANUE system. This was done to show the ability of the protocol to perform correctly regardless of distance.

The combination of a 1GiB read block size and 4 parallel connections could not be reliably tested for either TCP or RDMA. This is due to memory constraints of the test systems, and the memory requirements of *globus-url-copy* and *globus-gridftp-server* with this configuration.

There are several common themes that will be emphasized throughout the results. The block size used for the transfer played a significant role in the throughput attained. In general, with smaller block sizes TCP performs better, whereas RDMA performs better with larger block sizes. The exact cutoff of large vs. small depends on factors such as the number of streams, how much data was transferred, and the latency of the connection. Where CPU utilization is concerned the RDMA driver exhibited better performance in all

cases. CPU performance was affected by factors such as disk performance, the method of generating data, and of course the amount of data and the latency of the connection.

In summary, the following items were examined:

- Varied disk read/network write sizes to show impact on TCP and RDMA
- Varied transfer sizes to show impact of setup/teardown
- Varied Latency to show performance over long distances
- Disk vs. Memory Data sources and time limited transfers

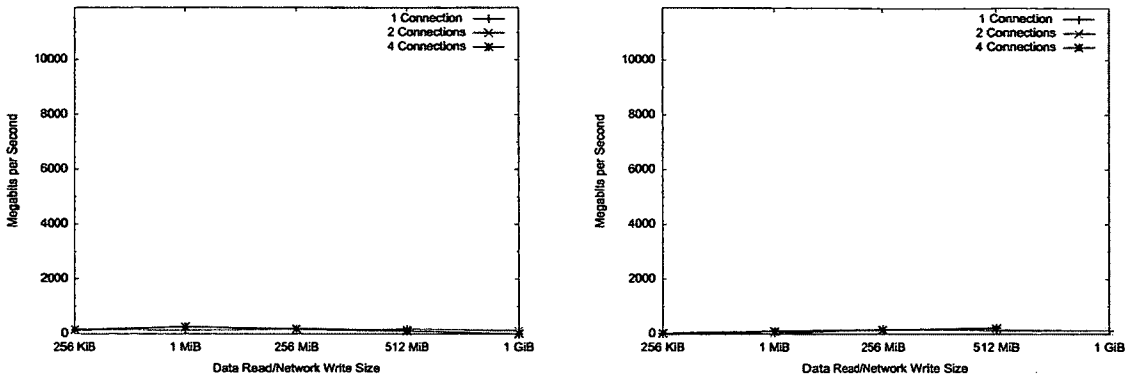
6.2 Read/Write Size and Parallel Transfers

These tests were performed with the two servers connected back-to-back through the ANUE with a 48ms delay introduced. In both the disk-to-disk and memory-to-memory transfers, 10GiB of data were transferred. In the case of the timed tests, transfers ran for a duration of 5 minutes.

When the source and sink of the data was the physical disk, the throughput was very low for both TCP and RDMA drivers. As shown in figs. 6-1a and 6-1b the size of the data unit read from disk, and subsequently written to the network had little effect on the overall throughput. In addition, the throughput itself was very low, much less than the line rate of 10Gb/s.

The throughput of the TCP driver when performing a memory-to-memory transfer is shown in fig. 6-2a. TCP maintained a relatively consistent throughput for all block sizes, and more streams generally performed better. With one connection (shown in red), TCP had an average throughput of 3357Mbps. Two parallel connections (shown in green) saw generally decreasing performance with increasing block size, and had an average throughput of 5497Mbps. The performance with 4 parallel transfers (shown in blue) also decreased slightly as block size increased, and had an average throughput of 5794Mbps.

In contrast, as show in fig. 6-2b, the performance of the RDMA driver was significantly affected by the block size. With smaller block sizes of 256KiB and 1MiB, the



(a) TCP (b) RDMA

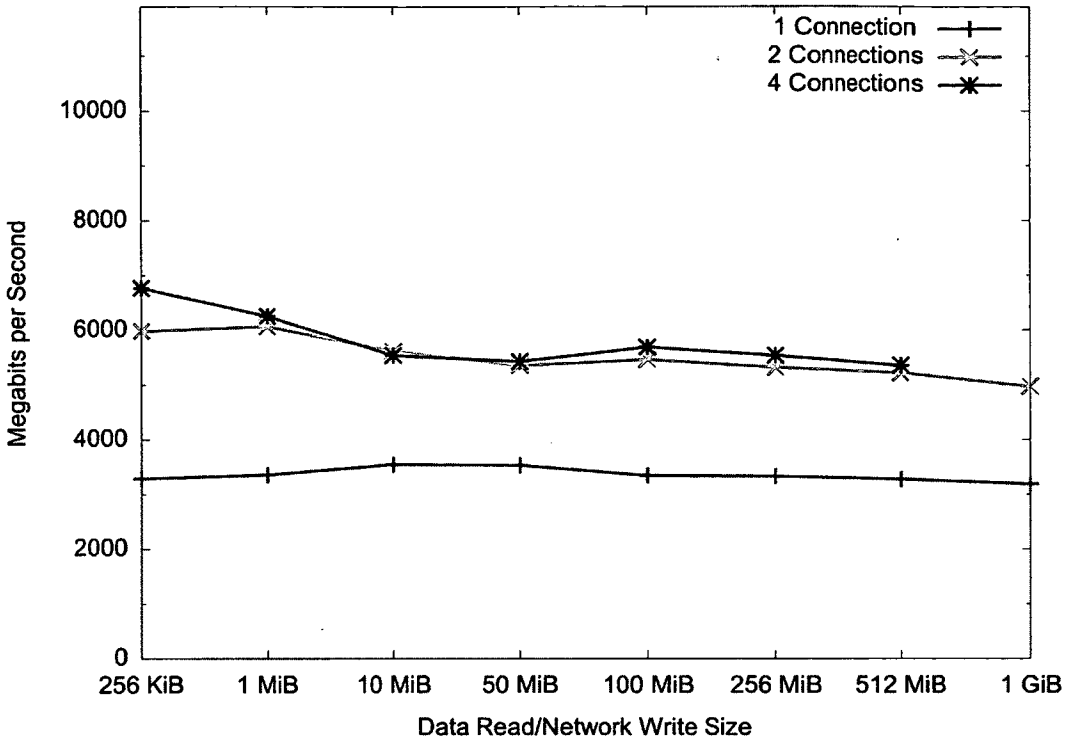
Figure 6-1: Throughput of 10GiB, Disk-Disk Transfer, 48ms Latency

throughput was negligible, only achieving 41Mbps throughput with 4 connections. RDMA throughput remained worse than TCP for block sizes less than 100MiB. For block sizes of 256MiB, 512MiB, and 1GiB, RDMA saw a significant increase with 1 connection, going from 1200Mbps at 50MiB, to around 4000Mbps at 256 MiB, and 6200Mbps with 1Gib blocks. With 2 and 4 parallel connections a similar increase was observed, with a maximum throughput of around 7000Mbps in both cases.

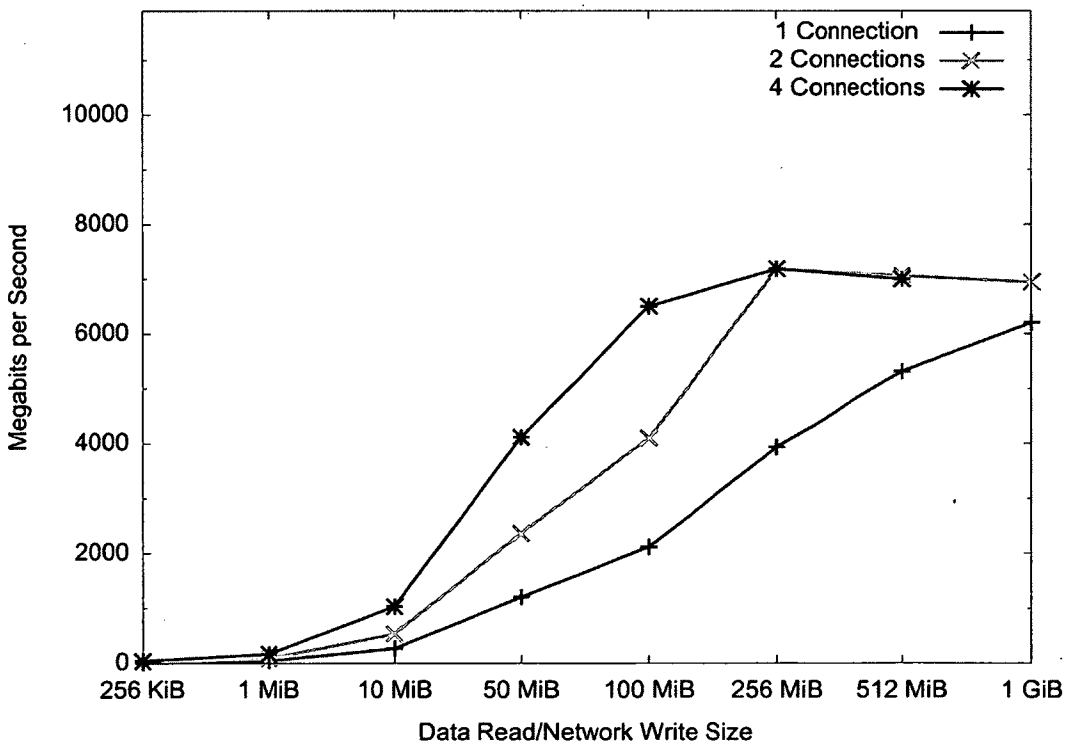
With time-limited transfers the TCP driver, as shown in fig. 6-3a, had consistent performance in all cases, with more connections resulting in higher overall throughput. A single connection saw a throughput of around 3500Mbps, adding another connection increased the throughput to around 6800Mbps, while 4 connections attained a throughput of around 9400Mbps. RDMA again exhibited a distinct improvement in throughput when the block size was increased, as fig. 6-3b shows. In all cases, 4 connections had higher throughput than 2 or 1 connections, and the maximum throughput was 9247Mbps, with 4 connections and 512MiB blocks.

6.3 CPU Utilization

The CPU Utilization data was collected at the same time as the throughput data discussed in the previous section.

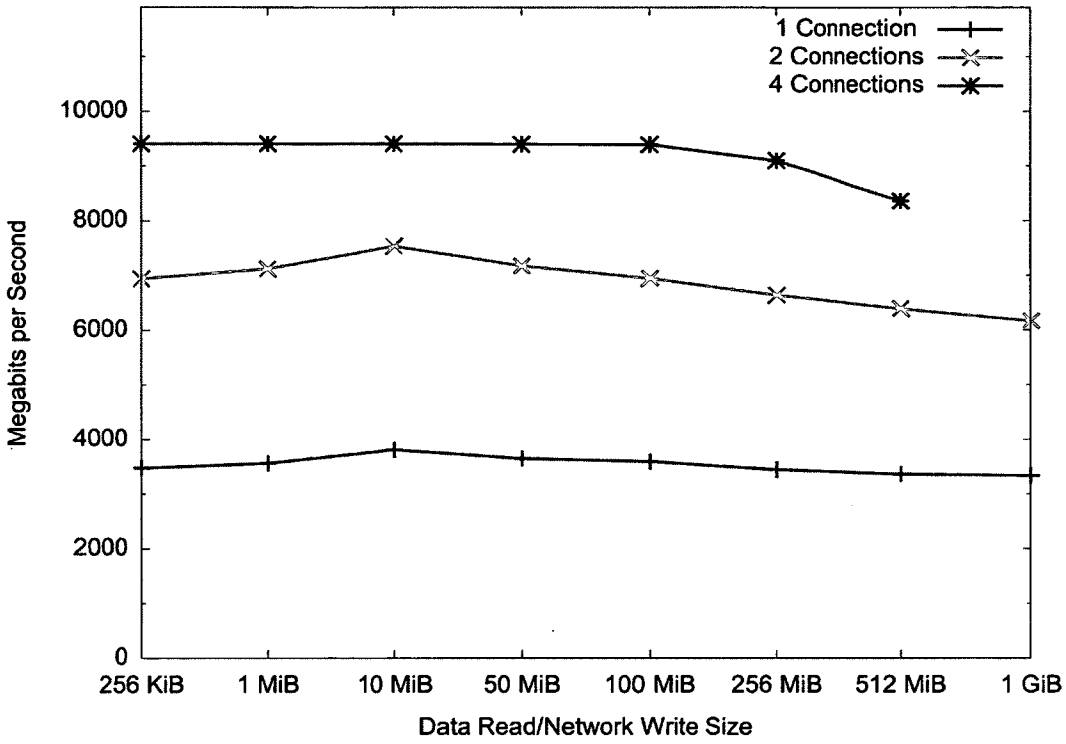


(a) TCP

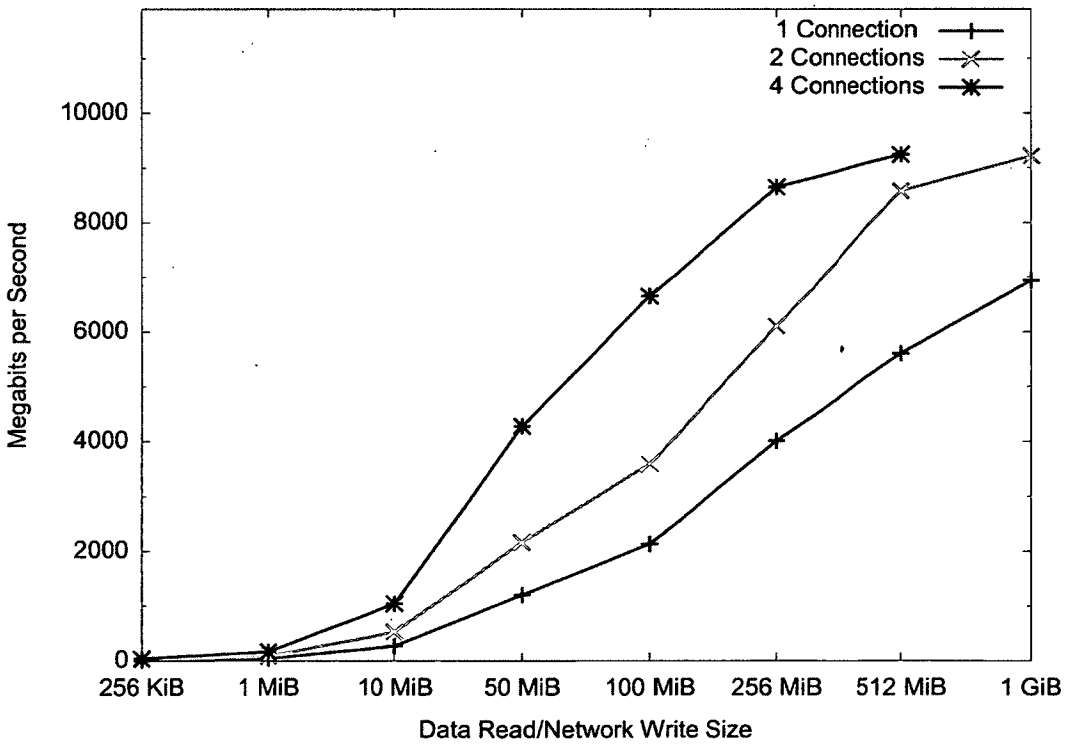


(b) RDMA

Figure 6-2: Throughput of 10GiB, Memory-Memory Transfers, 48ms Latency

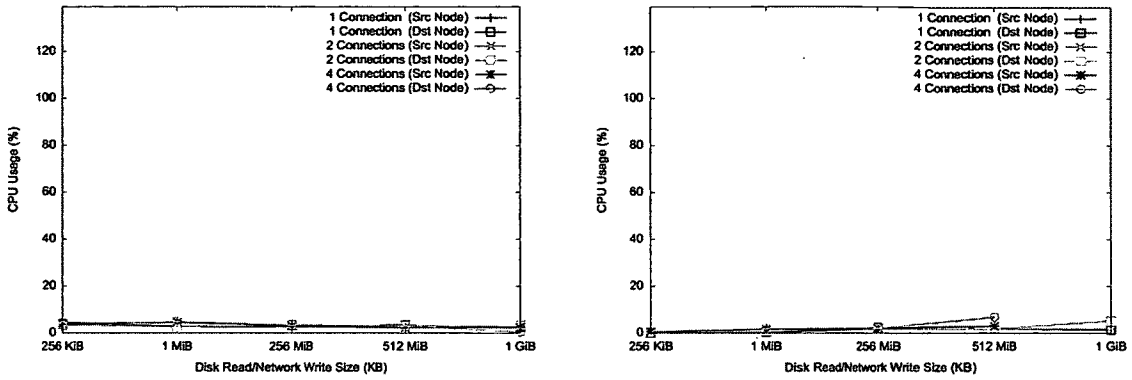


(a) TCP



(b) RDMA

Figure 6-3: Throughput of 5 minute Timed Memory-Memory Transfers, 48ms Latency



(a) TCP (b) RDMA

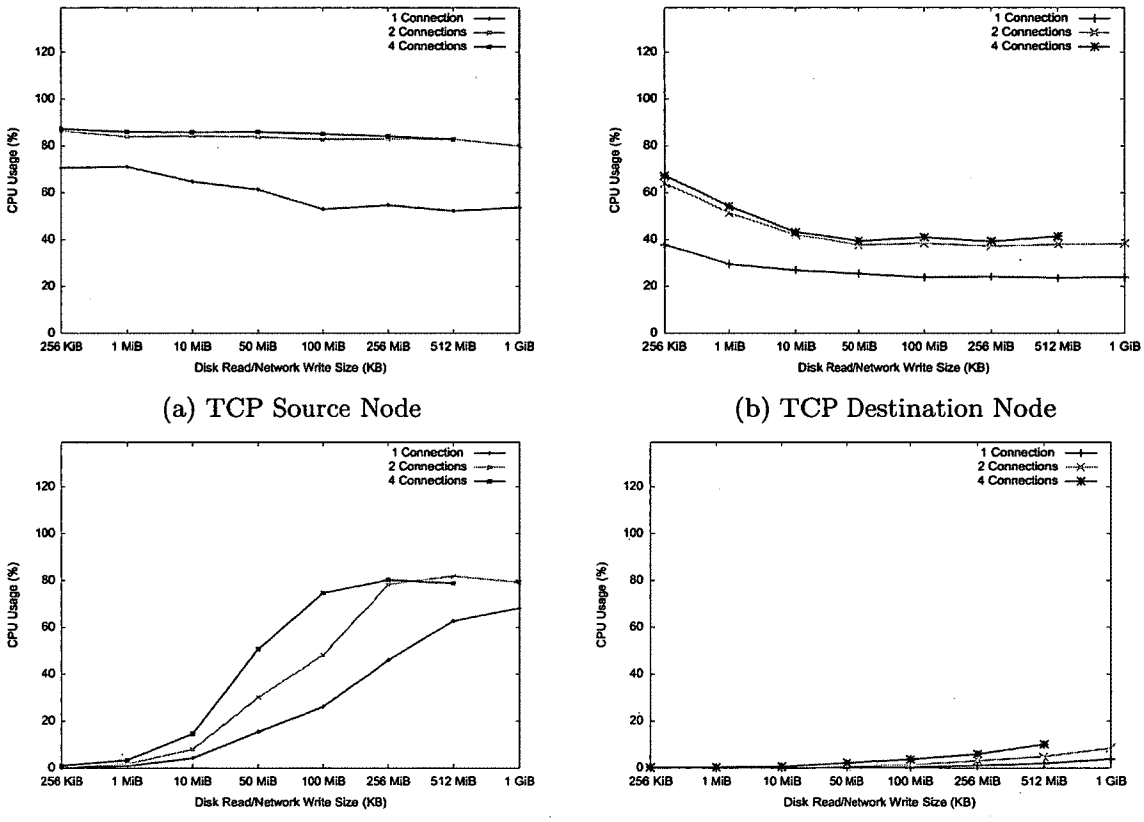
Figure 6-4: CPU Utilization of 10GiB, Disk-Disk Transfers, 48ms Latency

For both TCP and RDMA disk-to-disk transfers, shown in fig. 6-4, showed very low CPU utilization. This was true for both drivers in all test cases. The CPU utilization never exceeded 5%.

When data was transferred from memory, the effects of both drivers on CPU utilization are very apparent. Figure 6-5 shows CPU utilization for varying block sizes and number of connections, for both the source server and destination server with both drivers. With the TCP driver, when transfers were performed from memory to memory, CPU utilization increased dramatically on both source and destination servers from the disk transfers. The TCP driver used around twice as much CPU on the source server as on the destination server. With a single connection, the TCP driver used 55% to 70% of the CPU on the source server, and only 25% to 40% on the destination server. With 2 and 4 connections, the source server saw around 85% CPU utilization, while the destination server saw CPU utilization in the range of 40% to 65%.

The RDMA driver exhibited very different CPU usage on the source server when compared to the destination server. On the source server, with smaller block sizes the CPU utilization was minimal, but increased to 70% to 80% with the larger blocks. On the destination server, the RDMA driver did not exceed 5% utilization, except for the larger block sizes, where the CPU utilization was at or just below 10%.

The trends continue with the timed transfers, as fig. 6-6 shows. The TCP driver had



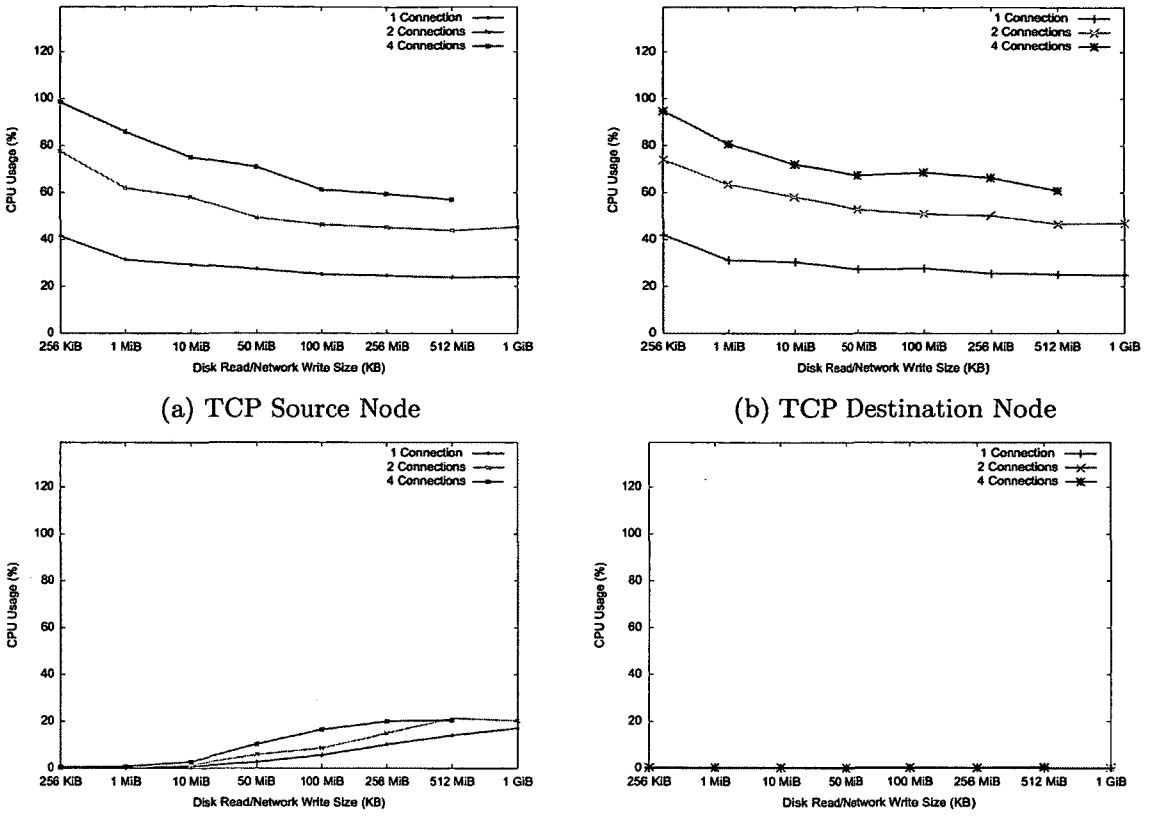
(a) TCP Source Node

(b) TCP Destination Node

(c) RDMA Source Node

(d) RDMA Destination Node

Figure 6-5: CPU Utilization of 10GiB, Memory-Memory Transfers, 48ms Latency



(a) TCP Source Node

(b) TCP Destination Node

(c) RDMA Source Node

(d) RDMA Destination Node

Figure 6-6: CPU Utilization of 5 minute Timed Memory-Memory Transfers, 48ms Latency

similar CPU usage on both source and destination servers. With a single connection, the CPU utilization went from 40% at the smallest block size, to around 25% at the largest. With 2 connections, the CPU usage nearly doubles, and with 4 connections the CPU utilization was near 100% for the smallest block sizes, while still using 75% at the larger sizes. The RDMA driver saw slightly higher CPU utilization on the source server than the destination server, utilizing at most 20% of the CPU. On the destination server, CPU utilization was consistently less than 1%. Overall, CPU usage for RDMA was significantly lower than TCP for all timed transfers.

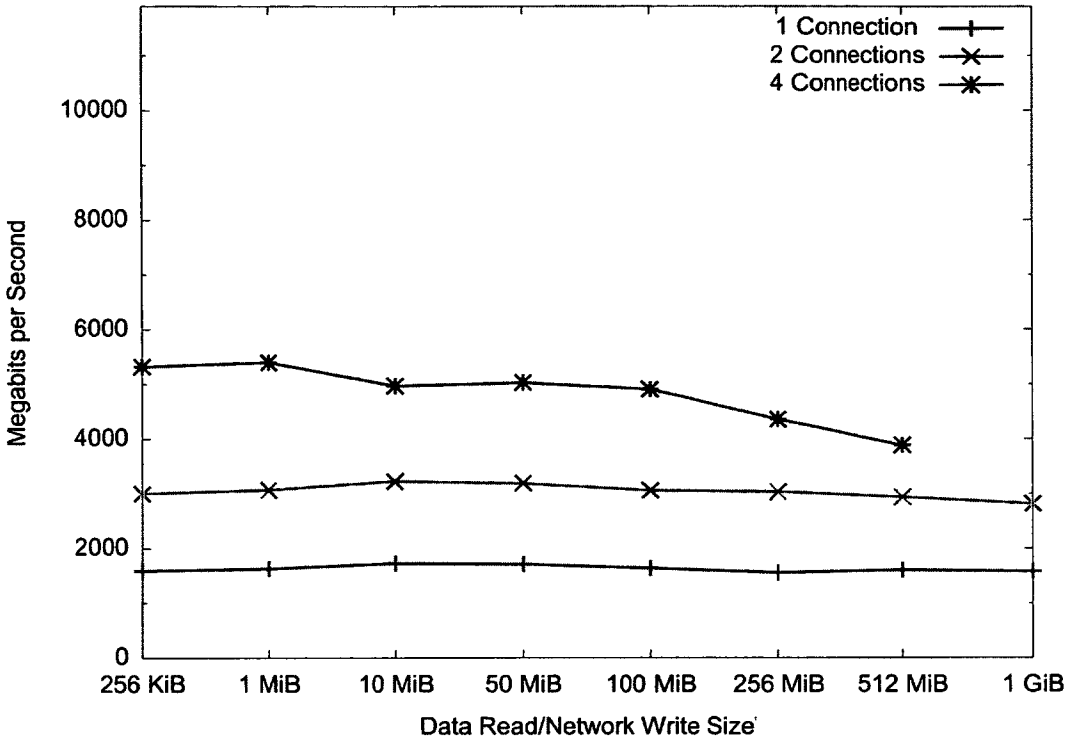
6.4 Longer Distance Transfers

For this set of tests, the delay on the line was extended to values of 100ms and 500 ms. 10GiB of data were transferred.

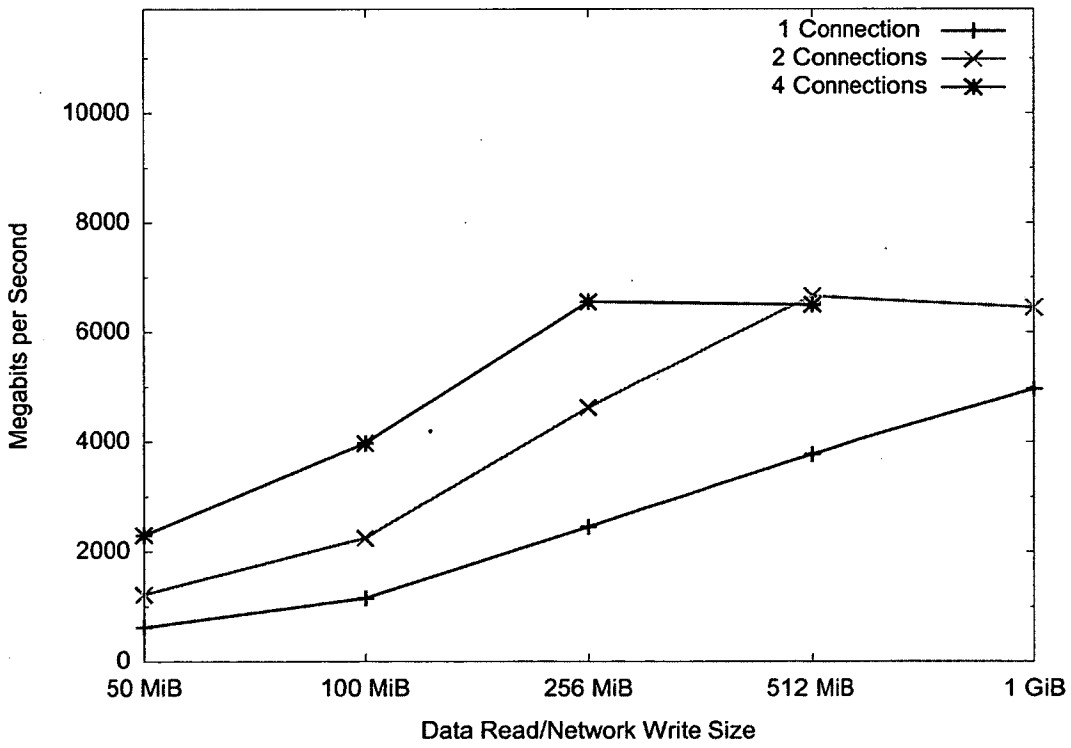
For both RDMA and TCP, tests for the 1GiB block size with 4 connections were omitted, because performance of these tests was generally as good, or worse as other block sizes, using either driver. Further, due to extremely low performance, tests of the smaller block sizes, 256KiB, 1MiB, and 10MiB, were omitted for RDMA.

Both the RDMA and TCP driver showed higher throughput at 100ms delay, shown in figs. 6-7a and 6-7b, than at 500ms delay, shown in figs. 6-8a and 6-8b. At 500ms delay, the throughput never exceeded 1000Mbps with the TCP driver. At 100ms delay, the TCP driver achieved an average of around 1600Mbps, 3000Mbps, and 5000Mbps, with 1, 2 and 4 connections. The RDMA driver also saw better throughput at 100ms, than at 500ms, and achieved a higher throughput than the TCP Driver for larger block sizes, reaching a maximum of around 6500Mbps at 100ms, and 3000Mbps at 500ms.

As shown in fig. 6-9, CPU utilization at 100ms latency followed a pattern similar to that of earlier tests. The RDMA driver utilized the CPU than TCP, and the source node utilized the CPU more than the destination node. With the TCP driver the source server saw TCP utilization in the range of around 30% to 80%, while with the RDMA driver, the source server saw usage between 10% and 50%, reaching near 80% in some cases. At the destination server, the CPU usage was generally between 20% and 30% for the TCP driver, while usually less than 5% for RDMA, and near 10% for some cases. At 500ms latency, shown in fig. 6-10, the CPU was utilized more infrequently than at 100ms latency for both RDMA and TCP drivers. The TCP driver usually used around 20% or less of the CPU, with the RDMA driver using up to 40% CPU on the source server, and not more than 5% on the destination server.

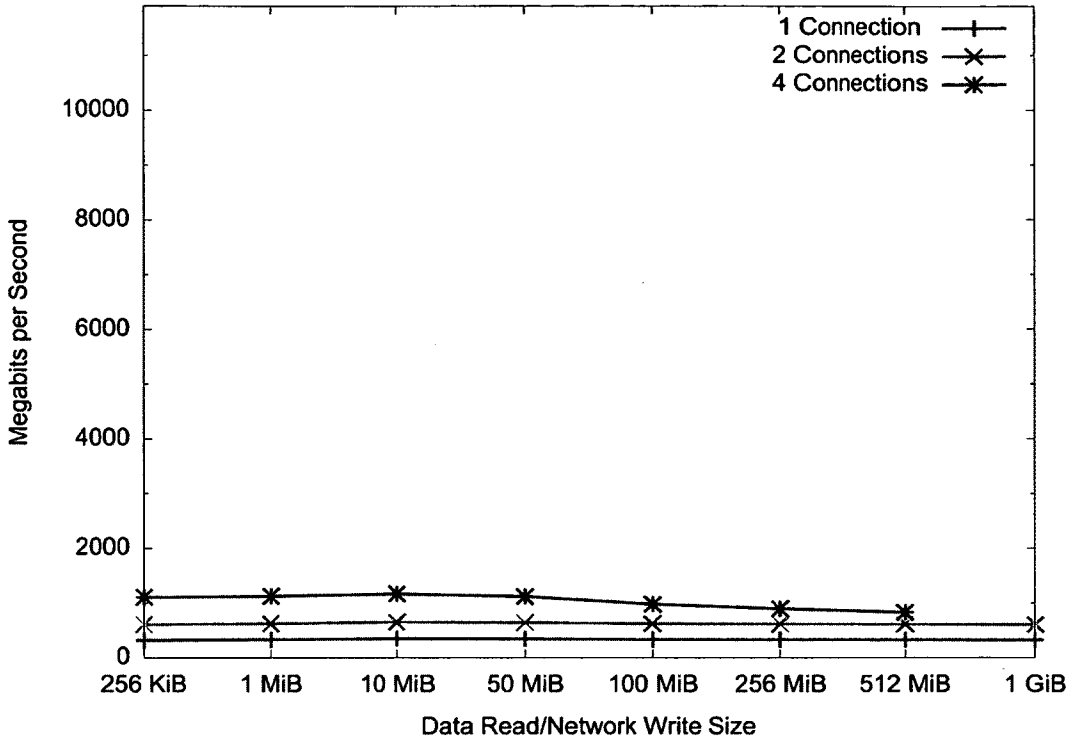


(a) TCP

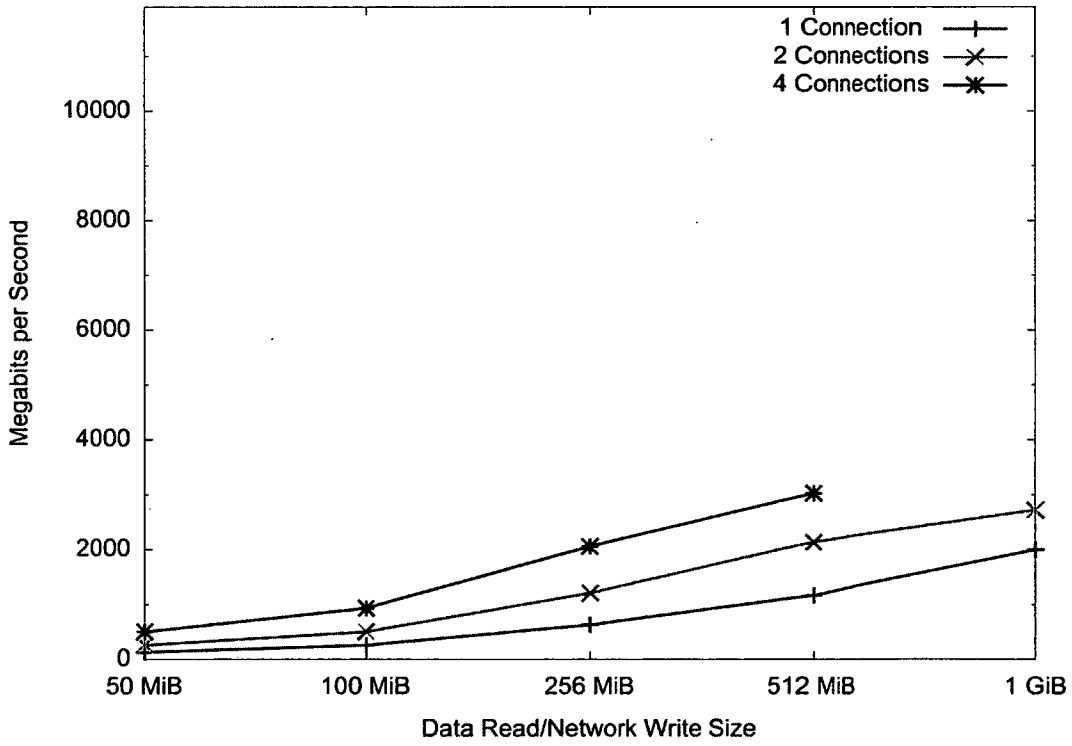


(b) RDMA

Figure 6-7: Throughput of 10GiB, Memory-Memory Transfers, 100ms latency

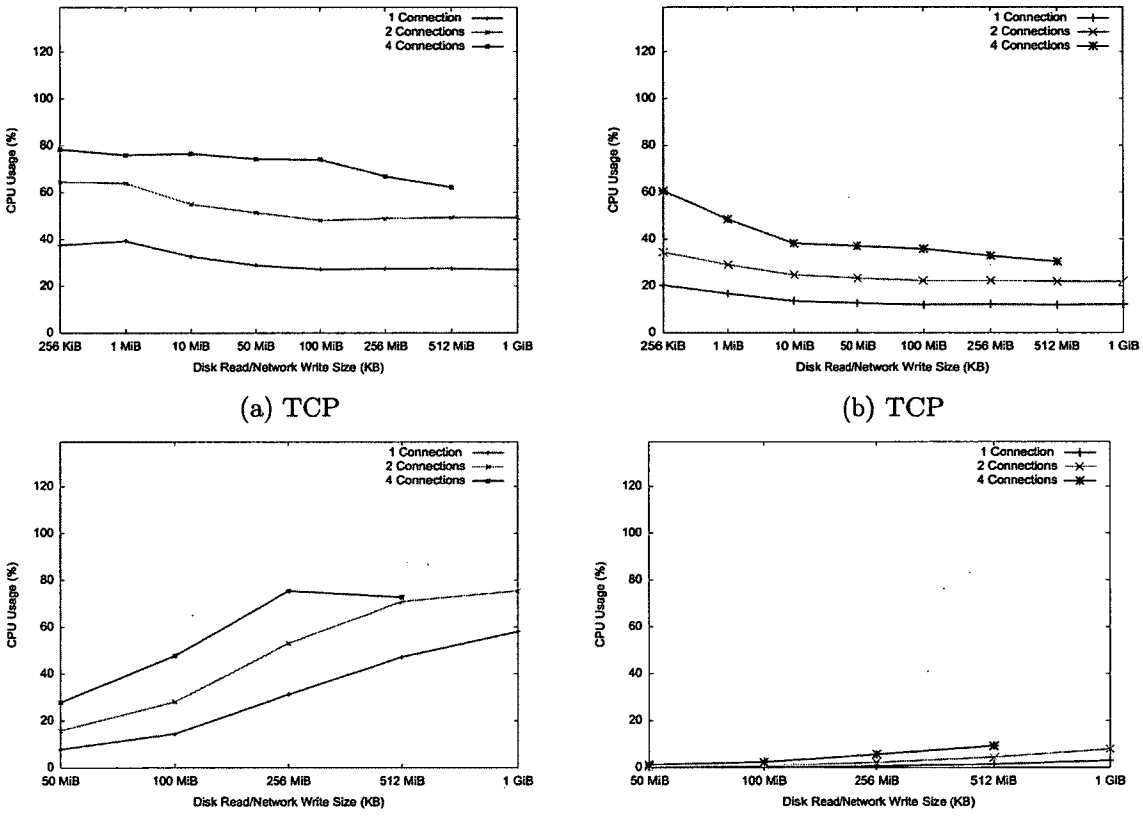


(a) TCP



(b) RDMA

Figure 6-8: Throughput of 10GiB, Memory-Memory Transfers, 500ms latency



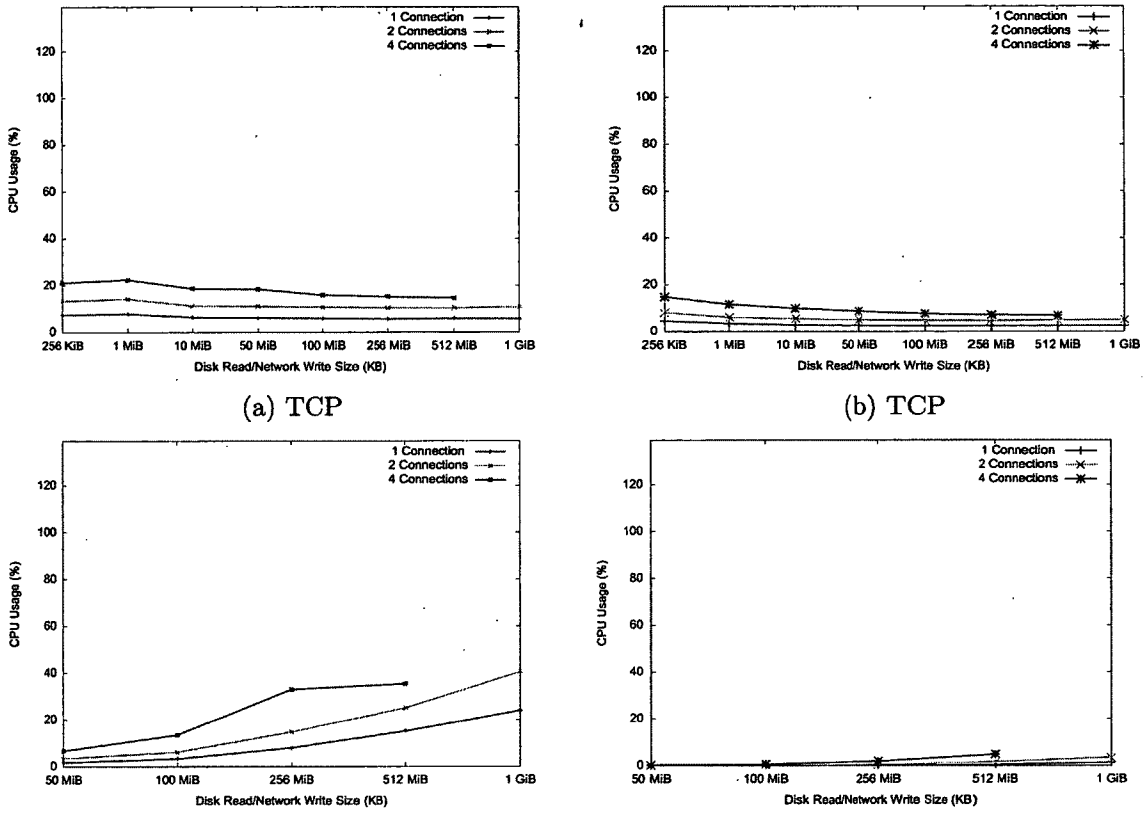
(a) TCP

(b) TCP

(c) RDMA

(d) RDMA

Figure 6-9: CPU Utilization of 10GiB, Memory-Memory Transfers, 100ms latency



(a) TCP

(b) TCP

(c) RDMA

(d) RDMA

Figure 6-10: CPU Utilization of 10GiB, Memory-Memory Transfers, 500ms latency

6.5 Very Large Transfers

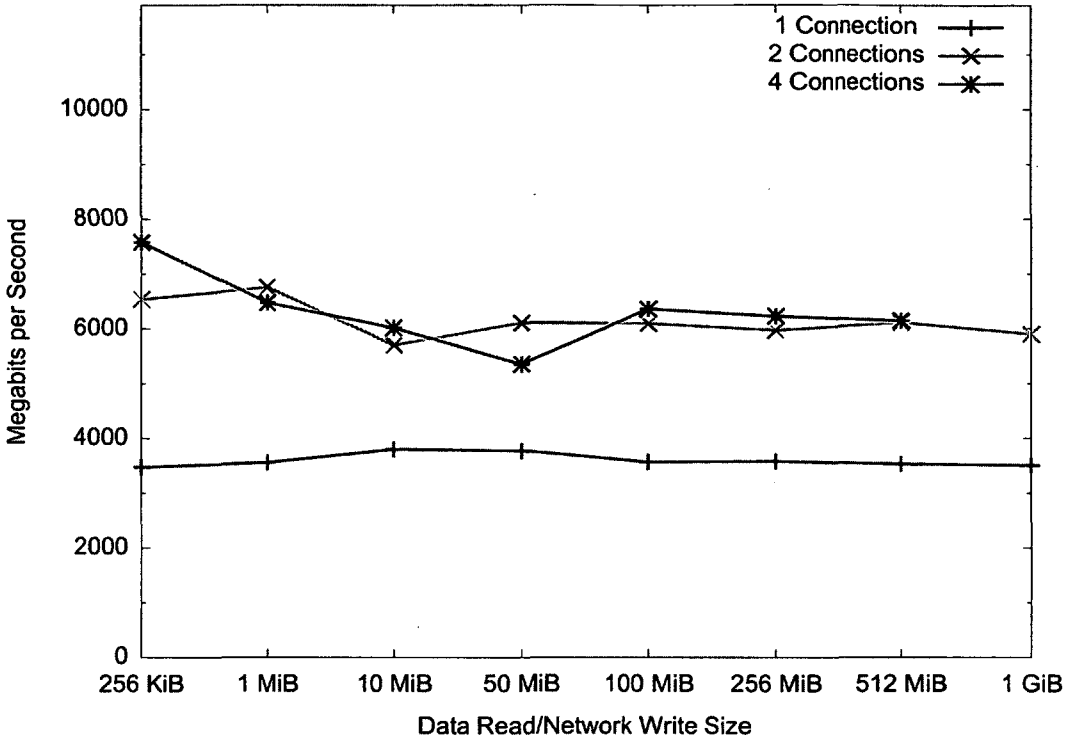
In these tests, data sizes of 100GiB and 500GiB were transferred from memory to memory, in the same fashion as earlier tests, with 48ms delay on the line.

As with the long distance tests, tests for the 1GiB block size were omitted for 4 connections due to memory constraints, as were tests of RDMA with 256KiB and 1MiB block sizes due to extremely low performance.

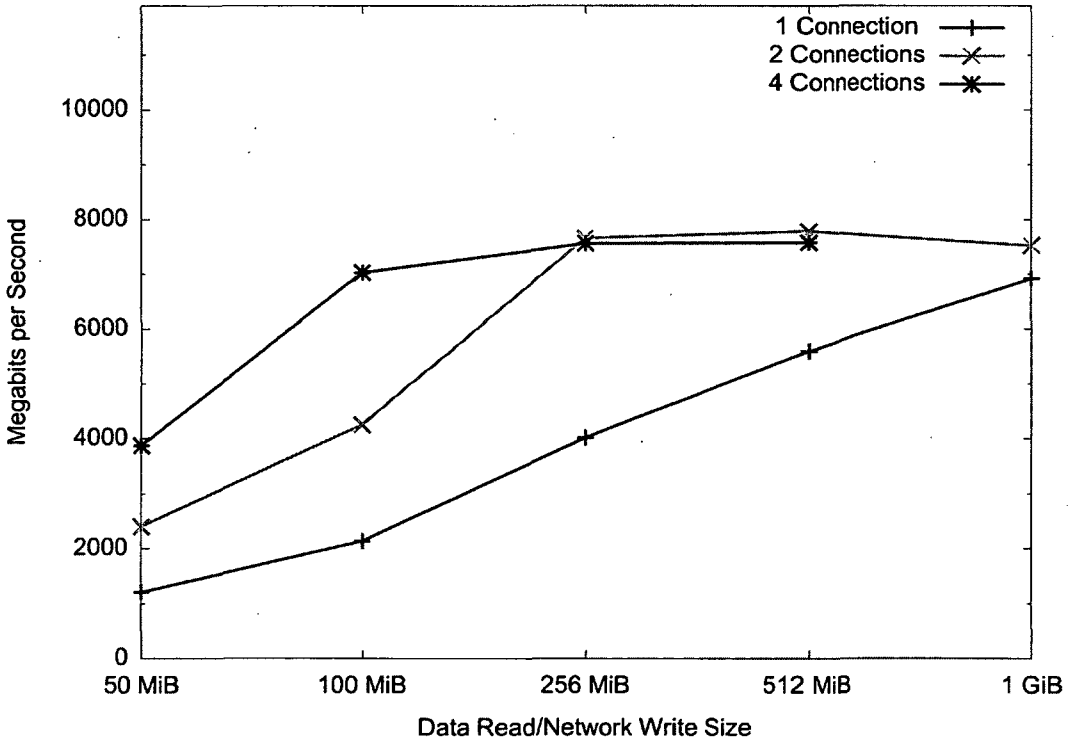
Figures 6-11a and 6-11b show that both RDMA and TCP drivers exhibited similar performance across all combinations, with RDMA performing marginally better. The TCP driver saw similar results for 2 and 4 connections, around 6000Mbps throughput. The RDMA driver saw throughput increase as the block size increased, reaching a maximum throughput near 8000Mbps. These values are similar, or slightly better for all block sizes and each stream size, when compared against the results from the smaller 10GiB data size, in figs. 6-2a and 6-2b.

Figures 6-12a and 6-12b show that the transfer of 500GiB of data showed comparable throughput, without significant difference from 100GiB, however RDMA was again marginally better at 256MiB and 512MiB. When 500Gib of data were transferred, the TCP driver achieved a throughput between 6000Mbps and 7000Mbps, while the RDMA driver achieved a throughput of nearly 8400Mbps for a block size of 1GiB.

Figures 6-13 and 6-14 show the CPU utilization for both driver types for 100GiB and 500GiB of data transferred. Similar trends from earlier tests are followed, and the results for both sizes are comparable. Again, the source server exhibited higher CPU utilization than did the destination server, and the TCP driver showed more overall usage than did the RDMA driver.

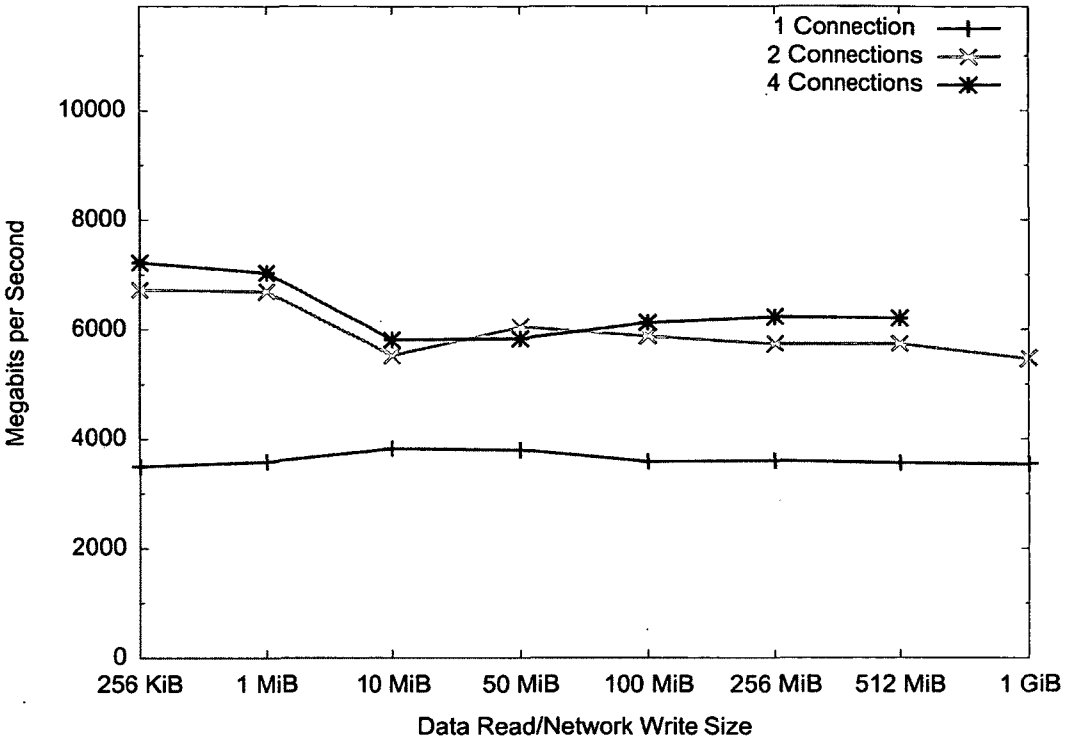


(a) TCP

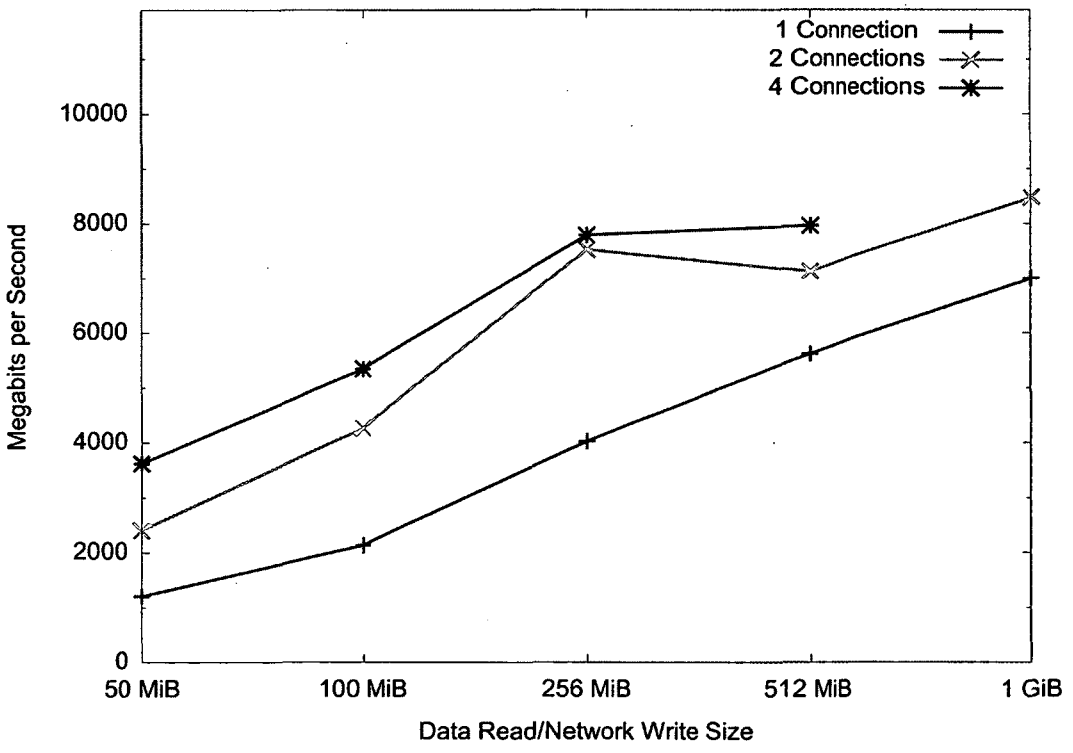


(b) RDMA

Figure 6-11: Throughput of 100GiB Memory-Memory Transfers, 48ms Latency

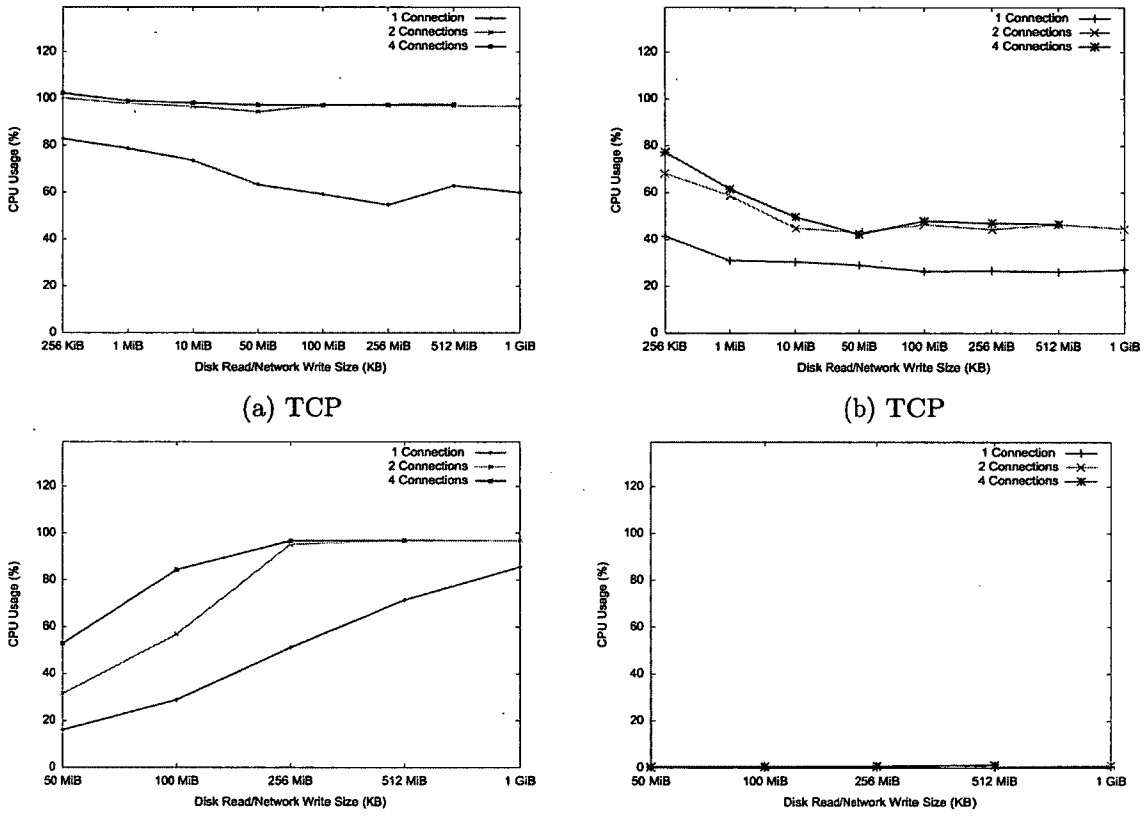


(a) TCP



(b) RDMA

Figure 6-12: Throughput of 500GiB Memory-Memory Transfers, 48ms Latency



(a) TCP

(b) TCP

(c) RDMA

(d) RDMA

Figure 6-13: CPU Utilization of 100GiB Memory-Memory Transfers, 48ms Latency

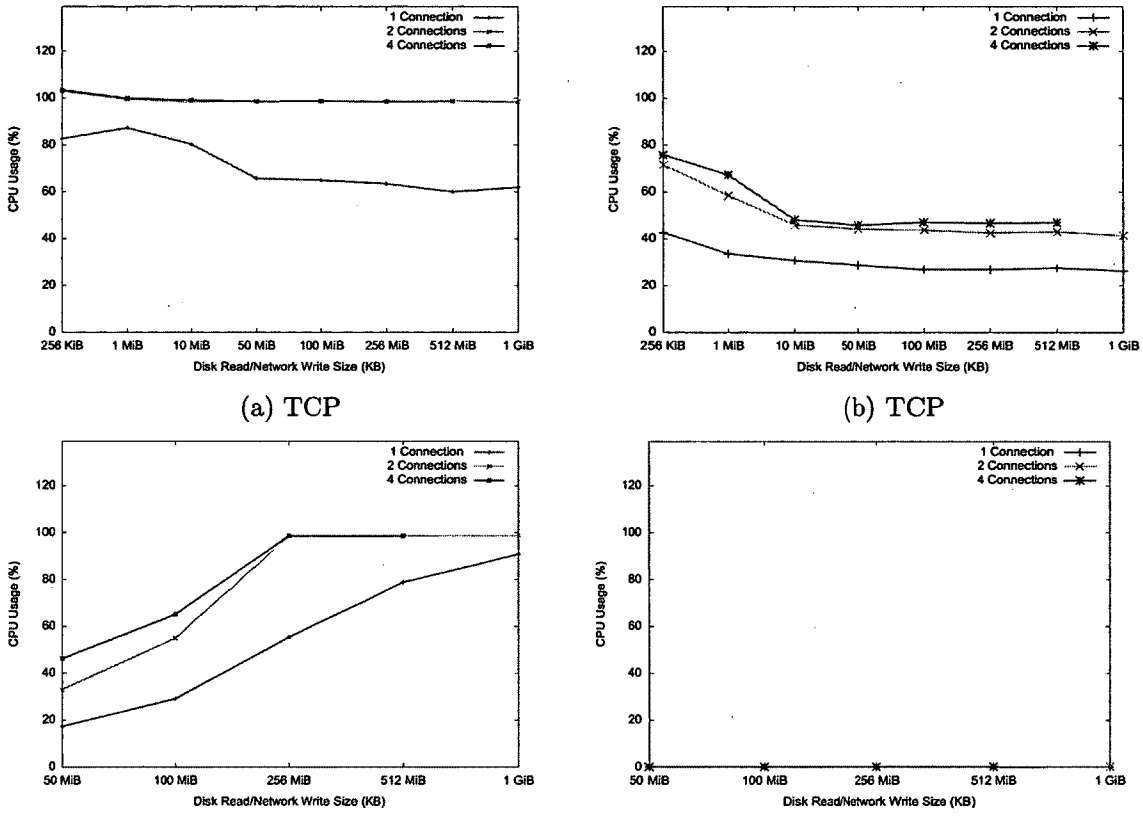


Figure 6-14: CPU Utilization of 500GiB Memory-Memory Transfers, 48ms Latency

CHAPTER 7

Analysis

This chapter provides an analysis for the results obtained, including recommendations for improving performance.

7.1 Analysis

7.1.1 Throughput

The RDMA driver implemented here experienced the same effects as other work [13] where message size was concerned. When using the *globus-gridftp-server* default of 256KiB message sizes, the RDMA Driver had very poor throughput in all transfer scenarios: disk-to-disk, memory-to-memory, and timed transfers. In fact, the RDMA Driver required message sizes of 256MiB, 1000 times larger, before throughput reached a level suitable for production use. This makes the default values an unrealistic choice for production systems.

The message size needed to obtain a high throughput can be mitigated by having a sufficient number of buffers waiting to be written to the network. With *globus-gridftp-server* and *globus-url-copy*, there is no way to affect the number of buffers given to a write operation. To the user, this means that tuning the message size with the "-bs" option according to the latency is the only avenue for improving performance with RDMA. This will limit the performance of the RDMA driver according to the amount of system memory available. In these experiments, the application was not able to sustain a high throughput for transfers using a 1GiB block size with more than two streams due to insufficient physical memory. With 10Gb/s line speed, near line-rate throughput could be achieved with timed transfers using smaller buffers of 256MiB and 512MiB, and also 1GiB with 2 parallel streams.

In order to scale for larger line speeds, or to increase performance with memory-memory, or disk transfers, larger message sizes will be a key for the RDMA driver.

7.1.2 CPU Utilization

As expected, the CPU utilization of RDMA was generally much less than that of TCP, with some exceptions. When the server hard-drive was used as the source and sink of the data, both drivers exhibited low CPU utilization. This can be attributed to time spent waiting for the drive during a read or write. During this time, the user program, and kernel are blocking waiting for the disk, and not utilizing the CPU. As these systems are not tuned as disk I/O nodes, memory and timed transfers were used to better evaluate CPU utilization. In the case of memory-to-memory transfers, the method used for generating data seemed to have an effect on the CPU. This can be shown by the difference in CPU usage for the source server when compared to the destination server. The source server had a much larger percentage of CPU utilization in the case of RDMA, and slightly more in the case of TCP. This can be attributed to using the *dd* utility for generating data. To limit the amount of data transferred, the GNU/Linux *dd* utility was used as an intermediary between the data source, "/dev/zero", and *globus-gridftp-server*. To do this, *dd* reads data from "/dev/zero" at constant 1KiB block sizes, up to a given number of bytes, and writes to a named FIFO, or pipe, created with the "mkfifo" command. The source server reads data from the FIFO and writes it to the network. The destination server reads data from the network, and writes directly to "/dev/null". In the case of TCP, even this write had significant CPU overhead, with RDMA the overhead was minimal, and often near zero.

7.1.3 Long Distance

Long distance transfers were simulated by increasing the delay on the line connecting the source and destination servers. With these transfers, run at 100ms and 500ms latency, the difference in throughput between the TCP and RDMA drivers was noticeable. This is possibly due to the TCP slow-start algorithm. TCP exponentially increases the amount of

data sent with each acknowledgement received. With longer delays, the acknowledgement takes longer to return, thus slowing the exponential growth. In contrast, RDMA works on a message basis, rather than on a stream, so the same amount of data is sent with each write. RDMA still must wait for the acknowledgment to be completed prior to sending more data using the same buffer. This is why improvements are seen when using larger buffer sizes, and parallel connections. In short, to "fill the pipe" more data buffers must be used to allow for enough outstanding operations to avoid waiting for an acknowledgement in order to transmit more data.

CPU utilization in these long distance tests followed the pattern previously established for 48ms delay. The only difference being that overall utilization was lower with both TCP and RDMA drivers, across each combination. This is likely due to more time blocking while waiting for an acknowledgement from the destination server.

7.1.4 Large Data

In these tests, transfers were done memory-to-memory with data sizes of 100GiB and 500GiB at 48ms latency. Throughput was generally the same for both TCP and RDMA drivers when compared to tests with 10 GiB at 48ms delay, across all combinations. This indicates that with these particular data sets, the cost of establishing a connection was not significant.

CPU utilization was higher in all cases for the TCP driver compared to the RDMA driver. The source server always used more CPU than the destination server in non-timed memory-to-memory transfers. This can be attributed to significantly more time spent generating data by the *dd* utility.

7.2 Improvements

There are several items in the GridFTP and RDMA Driver architecture that, with collaborative improvements, will likely result in better overall performance.

7.2.1 Buffer Size

As currently implemented, the size of the buffers used for Disk I/O, and the size of the buffers used for Network I/O, are currently controlled by the same parameter, the *globus-gridftp-server* "-bs" option. The default value of 256K is much too small to obtain a reasonable level of performance on RDMA for anything but the shortest of links. However, increasing this size alone is not an optimal approach to resolving this issue. Indeed, some servers may benefit from a particular Disk I/O buffer size, which is the justification for the tunable parameter. What is needed is another independent parameter to affect the size of the Network I/O buffers.

Ideally, this parameter should default to a much larger value with the RDMA Driver loaded. Additionally, this value should be identical on both the reading and writing side of the connection. Because RDMA operates on messages, and not on byte streams, the size of the receiving buffer must always be greater than or equal to the number of bytes sent or a hardware error will ensue, in lieu of inefficient protocols to handle the mismatch. Further, a GridFTP user may not have access to the source and destination servers to change the value of the block size, either for security, administrative, or other reasons. Therefore, the size of the Network I/O buffers should be a centrally agreed upon value, likely set by the user through an option to *globus-url-copy*, and communicated to the server(s).

There are at least two possible alternatives for how a Network I/O buffer size setting could be implemented.

1. Let the optimal size of the disk I/O buffer be D bytes. Also, let k be a coefficient such that $k > 0$ and $1 < k * D < 1000MB$ is the optimal network I/O buffer size. The 1000MB value is an arbitrary value for the largest message size to be sent by RDMA, and could be tuned according to the measured round-trip time. The sending program then performs k disk reads into the k parts of one network buffer per every 1 write of that whole buffer (i.e., all k pieces). This process is reversed on the receiving side, with the program performing k disk writes from the k parts of 1 buffer read from the network. With this method, the disk is filling/emptying the k pieces of one buffer in

parallel with the network emptying/filling the other buffer.

2. Again, let the optimal size of the disk I/O buffers be D bytes. In this case the sending program allocates $2 * N$ user-space buffers, each of size D bytes, where $N > 0$. The sending program will deliver N buffers for I/O at a time, allowing the network card to be transmitting at all times. Currently, GridFTP uses a double buffering approach that allows for only 1 buffer to be queued for transmission at a given time. With this alternate approach, the disk is filling/emptying N buffers in parallel while the card has enqueued the other N buffers for simultaneous transmission.

From the point of view of RDMA, the first alternative will likely provide the highest bandwidth utilization at lowest CPU cost, since there are fewer network I/O operations to move the same amount of data.

In the absence of this configuration, an improvement to the method for setting the mode-threshold could be made. By measuring the round-trip time during the open, the driver can determine what size buffer is appropriate to fill the link. This value could then be used with the expected data sized to determine how to proceed. For the purposes of testing the driver, when any buffer was greater than 1MiB, all were transferred using the registered mode.

7.2.2 Number of Buffers

The number of buffers issued in a single write operation is as important as the size of the individual buffers. In order to maximize throughput with RDMA, the necessary number of pending operations decreases with the size of the buffer in each operation. GridFTP does not have a parameter for specifying the number of buffers (i.e. the length of the *struct iovec[]* array) in each write operation. From observation, in each operation 1 data buffer is offered with each write, alternated with 1 other buffer. This allows 1 buffer to be filled while the other is being transmitted. It has been shown that for a connection with 48ms RTT, even with 1GiB message sizes, 2 writes must be outstanding to reach maximum throughput [13]. This number increases with the latency of the line.

7.2.3 Buffer Registration

Registering data buffers with an RDMA Channel Adapter is a required part of the "zero-copy" kernel by-pass data transfer process, and must be done before data may be transferred. The registration is done to allow the Channel Adapter to transfer data directly into, and out of application memory by obtaining a physical to virtual memory mapping. This is done by "pinning down" the memory region in virtual-memory, to prevent it from being paged out by the OS.

With the RDMA driver, registration happens "on-the-fly" as buffers are delivered to the I/O functions. To limit the effects of the high cost of the registration operation, memory registrations are cached, with the expectation that a buffer will be reused for data later in the transfer. At the conclusion of the transfer, memory regions are deregistered, as part of the closing of the connection. While this method of registering data is effective for transferring data, it has a number of drawbacks. First, the registration process itself is complicated, and logically separate from the work of the I/O functions. Additionally, the effectiveness of the registration cache in saving time is closely related to the choice of the function used in the caching scheme. With a small number of buffers, a linked-list approach is sufficient, however larger numbers of buffers will quickly render this scheme ineffective, requiring more complicated approaches. Regardless of the approach, this again is an increase in complexity not related to the I/O operation itself. Further, the driver itself has no notion of how many buffers the application will utilize, so the choice of algorithm is dependent on the worst case. Lastly, the operation of RDMA requires that the sending side of the connection know specific details about where data will be written on the receiving side. With the current approach, this requires an extra negotiation for every write operation, as the receiving side buffer cannot be predicted ahead of time.

One approach to solve the problem of registering buffers would be to provide a function to the user application to register a buffer prior to using it for a read or write. This could manifest itself in a couple of ways. One option would be to modify the driver API itself, with a driver specific buffer initialization function, that is available to the application. Another

option would be to embed this function as part of a shim during the buffer allocation process. This would make the process totally transparent to the user, and would allow the memory region to not only be allocated, but also registered with the Channel Adapter and driver, and aligned on a suitable memory boundary.

7.2.4 Algorithm Enhancements

To maximize the throughput and efficiency of the RDMA driver the number of protocol messages exchanged should be limited. The round-trip time incurred by these messages can quickly inhibit the effectiveness of the driver, particularly over long distances.

The GridFTP application is expected to be the primary application using the RDMA driver, which means it is especially important to make the protocol efficient for typical GridFTP operation. The primary example of this is with Mode-E operation. Mode-E, also known as EBLOCK (Extended block) is the key feature for enabling parallel connections and striped transfers. The Extended Block Header contains 3 Fields totalling 17 bytes: 1) an 8-bit Descriptor, 2) a 64-bit Byte Count, and 3) a 64-bit Offset Count. This header is sent ahead of every block of user data transferred. This allows for message reordering, marking, restart, and checkpointing for performance reporting to the user. [10]

The EBLOCK is an important consideration for the RDMA driver. Given the small size, it is an ideal candidate for utilizing the buffer-copy mode. When paired with a small data buffer the transfer proceeds simply with copying both buffers. However, when paired with a larger data buffer a hybrid approach could be taken by copying the EBLOCK and using buffer-register mode for the data buffer.

The protocol itself should be generic to maintain efficiency with any application, however, GridFTP is expected to be the primary user of the RDMA driver, so it is important to take into account typical GridFTP behavior. With a hybrid approach, in a single write operation some buffers could be transferred using copy-mode, while others use register mode. In this mode, rather than determine a mode to use for a given write operation, a mode is chosen per buffer. This means that in the GridFTP case, the EBLOCK may be copied,

while the succeeding data buffer may be registered. This requires a more generic protocol. This protocol would reuse the concept of an initial "signal" message to indicate the mode to the receiver. The improvement would be to include any user data that was copied with this signal message, and to indicate to the reader where to find the data. This eliminates the need for a return message from the reading node. In the case of a registered transfer, the signal message would again be used, but the receiver would still send a message indicating where the data should be written. The improvement would come from the ability to transfer the EBLOCK in the signaling message, with the user data to follow, which is expected to increase the performance of the driver.

Lastly, in the copy-mode, data can be transferred immediately, without any signaling from the reader. Along with the data, the writer can use the immediate data field of the RDMA_WRITE_WITH_IMM to indicate to the reader in what index of the private buffer space the data may be found. Knowing the number of available vs. allocated buffers allows the writer to stop before it has overflowed the receive buffers. All that is required is an occasional acknowledgement from the reader, to indicate how many free buffers are available.

CHAPTER 8

Conclusion

This chapter reviews the work and many of the notable findings. In addition, avenues for further work and future exploration are noted.

8.1 Review

This work demonstrates the viability of RDMA, and specifically RoCE, as an underlying transport method for the GridFTP application. The performance of the RDMA Driver can be broken down generally into two cases, those with sufficiently sized messages, and those where the size of the message was insufficient to maintain performance.

In cases where the size of the message was 256MiB or larger, the RDMA driver performed as well, or better than the TCP driver, both in regards to throughput and CPU utilization. In memory-to-memory, and timed transfers, as the size of the message increased, RDMA throughput also increased, demonstrating the importance of this feature of RDMA. TCP in contrast was generally unaffected, or slightly negatively affected by block size.

CPU utilization was significantly higher with the TCP driver. In fact, generally the RDMA driver had near zero CPU usage, when the method of generating data for transmission is taken into account. This is an important finding, as it shows that with current hardware, the RDMA transport will scale with increases in the speed of the network. Further, with today's network speed, this allows data transfer servers to perform other tasks while a transfer is ongoing, or perhaps utilize other RDMA hardware for other transfers. TCP however, as shown in these experiments, will be limited in many cases as CPU utilization is already near 100%.

Unsurprisingly, both RDMA and TCP driver's throughput suffered due to poor disk I/O performance. This makes it particularly important to optimize nodes that will be used for transferring data not only for network speed and efficiency, but also for disk I/O performance.

8.2 Future Work

8.2.1 Generic Driver Enhancements

As with any piece of software, there are many enhancements that can be made to the RDMA driver to improve all around performance. Several of the most pressing enhancements are itemized below.

- **Robustness:** RDMA requires several more steps to organize and prepare for data transmission than traditional sockets. The establishing of a connection, setup of the queue-pairs, and submitting work requests and processing completions are all elements that can experience failures. The ability of the driver to catch errors, interpret them, and signal the nature of the error to the XIO subsystem, and the user, and to possibly recover from the error, are all elements that will improve overall operation.

- **Stability:** The GridFTP server application, in many cases, oversees many transfers without restarting. The RDMA driver has been shown to be reliable in many different cases, however continued testing in a variety of situations should be done to ensure the RDMA driver does not cause a larger failure of the GridFTP server application.

- **Efficiency:** The RDMA driver implementation has been developed with an eye towards efficiency, but with the primary goal of correctness in mind. As updates are made for robustness and stability, more work should be done to ensure all operations are as efficient as possible, especially if any updates to the GridFTP application are made.

- **Clarity and Simplicity:** The transfer semantics of RDMA, while powerful and flexible, are not always straightforward in the implementation. This is especially true for those not familiar with RDMA verbs. To encourage participation in the development process across the Globus XIO community, an effort should be made to document the code, both inline and on paper, and to simplify coding structures where ever possible.

There are a number of areas where future exploration and development is warranted. Primarily, as shown in this work, there are a number of upgrades that can be made to the *globus-url-copy* and *globus-gridftp-server* applications to increase RDMA performance. The first of these is a parameter to allow the network write block size to be set via the command line of *globus-url-copy*. This will remove the necessity of setting the disk read size for the server application for the purpose of increasing RDMA throughput, and will allow for more configuration options for the administrator. In addition, a method for describing to GridFTP how many operations can be given to a driver I/O function will allow for RDMA to deliver multiple data buffers to RDMA hardware with each call. This will again allow more flexibility in attaining high throughput. Lastly, as RDMA depends on data buffer memory registration for zero-copy data transfers, implementing a method to allow GridFTP applications to register data buffers prior to use will significantly improve the performance of the RDMA driver, as well as simplify the operation of the driver, and the code itself.

BIBLIOGRAPHY

[1] Bay Microsystems. http://www.baymicrosystems.com/solutions_data_center_infiniband.php.

[2] globus online. https://www.globusonline.org/.

[3] Globus Toolkit. http://www.globus.org/toolkit/about.html.

[4] GridFTP for Users. http://www.mcs.anl.gov/~kettimut/tutorials/GridFTP4Users_Argonne09.pdf.

[5] Lustre File System. www.lustre.org.

[6] Obsidian Strategies. http://www.obsidianresearch.com/products/longbow/index.html.

[7] InfiniBand Architecture Specification Volume 1, Release 1.2.1. InfiniBand Trade Association, November 2007.

[8] Supplement to InfiniBand Architecture Specification Volume 1, Release 1.2.1: Annex A16: RDMA over Converged Ethernet (RoCE). InfiniBand Trade Association, April 2010.

[9] OFA Overview. https://beany.openfabrics.org/home/ofa-overview.html, May 2012.

[10] William Allcock, Joe Bester, John Bresnahan, Sam Meder, P. Plaszczak, and Steve Tuecke. GridFTP: Protocol Extensions to FTP for the Grid. *Security*, (August), 2003.

[11] William Allcock, John Bresnahan, Rajkumar Kettimuthu, and Joseph Link. The globus extensible input/output system (xio): A protocol independent io system for the grid.

In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 4 - Volume 05*, IPDPS '05, pages 179.1–, Washington, DC, USA, 2005. IEEE Computer Society.

[12] William Allcock, John Bresnahan, Rajkumar Kettimuthu, Michael Link, Catalin Dumitrescu, Ioan Raicu, and Ian Foster. The globus striped gridftp framework and server. In *In SC 05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, 2005.

[13] Timothy Carlin and Robert D. Russell. Experiments with RoCE on the ANI 100 GigE Testbed. May 2012.

[14] Steven Carter, Makia Minich, and Nageswara S. V. Rao. Experimental evaluation of infiniband transport over local- and wide-area networks. In *Proceedings of the 2007 spring simulation multiconference - Volume 2*, SpringSim '07, pages 419–426, San Diego, CA, USA, 2007. Society for Computer Simulation International.

[15] Ian Foster. Globus toolkit version 4: software for service-oriented systems. In *Proceedings of the 2005 IFIP international conference on Network and Parallel Computing*, NPC'05, pages 2–13, Berlin, Heidelberg, 2005. Springer-Verlag.

[16] Foster, Ian. What is the Grid? A three Point Checklist. Technical report, July 2002.

[17] Paul Grun. Introduction to InfiniBand for End Users. Technical report, April 2010.

[18] Paul Grun. Writing Application Programs for RDMA using OFA Software, January 2012.

[19] E. Kissel and M. Swany. Evaluating high performance data transfer with rdma-based protocols in wide-area networks. In *Proceedings of the 14th IEEE International Conference on High Performance Computing and Communications (HPCC-12)*, 2012.

[20] Ezra Kissel, Martin Swany, and Aaron Brown. Phoebus: A system for high throughput data movement. *Journal of Parallel and Distributed Computing*, 71(2):266 – 279, 2011. ¡ce:title¿Data Intensive Computing¡/ce:title¿.

[21] Ping Lai, Hari Subramoni, Sundeep Narravula, Amit Mamidala, and Dhabaleswar K. Panda. Designing Efficient FTP Mechanisms for High Performance Data-Transfer over InfiniBand. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, pages 156–163, Washington, DC, USA, 2009. IEEE Computer Society.

[22] Patrick MacArthur and Robert D. Russell. A Performance Study to Guide RDMA Programming Decisions. In *2012 IEEE 14th International Conference on High Performance Computing and Communications*, HPCC '12, May 2012.

[23] J. Postel. User Datagram Protocol. RFC 768 (Standard), August 1980.

[24] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[25] J. Postel and J. Reynolds. File Transfer Protocol. RFC 959 (Standard), October 1985. Updated by RFCs 2228, 2640, 2773, 3659, 5797.

[26] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia. A Remote Direct Memory Access Protocol Specification. RFC 5040 (Proposed Standard), October 2007.

[27] Sabine Richling, Heinz Kredel, Steffen Hau, and Hans-Günther Kruse. A long-distance infiniband interconnection between two clusters in production use. In *State of the Practice Reports*, SC '11, pages 15:1–15:8, New York, NY, USA, 2011. ACM.

[28] H. Shah, J. Pinkerton, R. Recio, and P. Culley. Direct Data Placement over Reliable Transports. RFC 5041 (Proposed Standard), October 2007.

[29] Hari Subramoni, Ping Lai, Raj Kettimuthu, and Dhabaleswar K. Panda. High performance data transfer in grid environment using gridftp over infiniband. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 557–564, Washington, DC, USA, 2010. IEEE Computer Society.

[30] Yuan Tian, Weikuan Yu, and Jeffrey S. Vetter. Rxio: Design and implementation of high performance rdma-capable gridftp. *Comput. Electr. Eng.*, 38(3):772–784, May 2012.

[31] Brian Tierney, Ezra Kissel, Martin Swany, and Eric Pouyoul. "efficient data transfer protocols for big data". In *Proceedings of the 8th International Conference on eScience*, IEEE, October 2012.

[32] Veeraraghaven, Malathi and Russell, Robert. SDCI Net: An integrated study of data-center networking and 100 GigE wide-area networking in support of distributed scientic computing. Technical report, September 2011.